Soft Warehouse

Handcrafted Software for the Mind

Reference Manual

muLISP™ 90

LISP Language Programming Environment

Soft Warehouse

Handcrafted Software for the Mind

Reference Manual

muLISP™ 90

LISP Language Programming Environment

Preface

The LISP computer programming language is based on a paper by John McCarthy entitled **Recursive Functions of Symbolic Expressions and Their Computation by Machine** [1960]. Since its first implementation at the Massachusetts Institute of Technology in the early 1960's, LISP has remained the "machine language" of the **artificial intelligence** (AI) community.

Going beyond its AI roots, LISP has evolved into a powerful general-purpose computer language. It is *the language of choice* for implementing state-of-the-art software because:

LISP is an *applicative* language that encourages a functional approach to designing large software systems.

LISP is a *recursive* language that makes it ideal for realizing complex mathematical algorithms.

LISP is a *list processing* language that can solve real-world problems by making transformations on lists or trees that model the problems.

LISP is an *extensible* language to which you can add your own functions and control constructs.

LISP uses *dynamic memory management* to automatically recycle and reallocate memory according to the demands of programs.

LISP is a *data-driven* language that allows a data structure to be executed as a program.

LISP is an *incremental* language that supports top-down, bottom-up, *and* middle-out programming styles as appropriate.

LISP is an *interactive* language that provides the software developer with a powerful programming environment.

For these reasons LISP is ideal for projects in such diverse areas as natural language translation, robotics, game playing, expert systems, theorem proving, compiler writing, systems programming, number theory, medical diagnosis, and program verification.

The current version of **muLISP** is the culmination of over twelve years of effort into designing a high performance, general purpose LISP

system for personal computers. While incorporating many of the ideas of modern mainframe systems, especially **Common LISP** (see Steele, [1984]), **muLISP** remains a *lean and mean* LISP system. The design philosophy behind **muLISP** is to provide the software developer with an efficient, compact kernel of functions and special forms defined in machine language, leaving sufficient space for sophisticated programming environments and large application programs.

As might be expected, some degree of study and patience is required to make maximum use of **muLISP** as a software development tool. The best way is through the *hands-on* experience provided by the Tutorial System (see Section 3.2). This on-line interactive system teaches the *fundamentals* of LISP programming. Chapter 10 lists many good LISP books and periodicals that cover more *advanced* programming techniques.

This manual is meant to serve as a reference guide for **muLISP**. Every effort has been made to make the information presented here as clear, comprehensive, and accessible as possible. However, this manual is *not* a tutorial on **muLISP** or LISP.

muLISP was conceived and implemented by Albert Rich of Soft Warehouse, Inc. IRRATNAL.LSP was written by David Stoutemyer of Soft Warehouse, Inc. FLAVORS.LSP and CLOSURE.LSP were written by Peter Ohler. EIGHTS.LSP was written by Peter Harada.

The design of **muLISP** was influenced by the suggestions of Richard Zipple, David Stoutemyer, Samuel Rhoads, Peter van Lith, Peter Harada, Martin Griss, Ralph Freese, Peter Deutsch, and many others. The clarity and completeness of this manual are due in large part to the patience and persistence of Joan Rich and David Stoutemyer.

Albert D. Rich
Applied Logician
Soft Warehouse, Inc.

Table of Contents

Title Page i
Copyright Notice ii
Preface iii
Table of Contents v

Chapter 1. Introduction

1.1 Major Features 2
1.1.1 Machine Language Kernel 2
1.1.2 Program Development System 3
1.1.3 Source File Library 4
1.1.4 Demonstration Game Programs 5
1.1.5 Tutorial System and Documentation 5
1.2 **muLISP** Diskette Files 6
1.3 System Requirements 8
1.3.1 IBM PC Compatible Computers 8
1.3.2 NEC PC-9801 Compatible Computers 8
1.3.3 Other MS-DOS Computers 8
1.4 Making A Working Copy 9
1.4.1 Hard Disk Procedure 9
1.4.2 Diskette Procedure 10
1.5 Keyboard Notation 11

Chapter 2. Fundamentals

2.1 Starting **muLISP** 13
2.2 Read-eval-print Loop 14
2.3 Line Editing 15
2.4 Programming 17
2.5 Source Files 19
2.6 Memory Image Files 20

Chapter 3. Learning muLISP

3.1 Demonstration Programs 23
3.2 Tutorial System 25

Chapter 4. Interpreter

4.1 Data Objects 27
4.1.1 Symbols 28
4.1.2 Numbers 29
4.1.3 Conses 30
4.2 Memory Management 31
4.2.1 Data Spaces 31
4.2.2 Garbage Collection 32
4.2.3 Data Space Reallocation 32
4.3 Break Option Menu 33
4.4 Console Interrupts 34
4.5 Error Messages 34

Chapter 5. Primitives

5.1 Predicates 40
5.1.1 Recognizers 40
5.1.2 Comparators 45
5.1.3 Logical Operators 50
5.2 Lists 53
5.2.1 Selectors 53
5.2.2 Constructors 60
5.2.3 Modifiers 70
5.2.4 Deletion 77
5.2.5 Search 80
5.2.6 Substitution 85
5.2.7 Sets 89
5.2.8 Association Lists 94
5.2.9 Sorting 98
5.3 Symbols 101
5.3.1 Value 101
5.3.2 Property List 108
5.3.3 Definition 112
5.4 Numbers 121
5.4.1 Precision 122
5.4.2 Numerical Predicates 125
5.4.3 Arithmetic Operators 131
5.4.4 Component Extraction 138
5.4.5 Bit-wise Logical Operators 144
5.4.6 Random Numbers 148
5.5 Print Names 151
5.5.1 Character Functions 151

5.5.2 String Functions 158
5.5.3 Symbol Creation 168
5.6 Control Constructs 169
5.6.1 Sequencing 169
5.6.2 Conditionals 173
5.6.3 Iteration 176
5.6.4 Non-local Exits 179
5.6.5 Subprocess Invocation 183
5.7 Interpreter 184
5.7.1 Form Evaluation 184
5.7.2 Function Application 189
5.7.3 Mapping Functions 191
5.7.4 Mapping Predicates 194
5.7.5 Macro Expansion 197
5.7.6 Memory Management 200
5.8 Communication 206
5.8.1 Input Functions 206
5.8.2 Read macro Characters 214
5.8.3 Input Control Variables 218
5.8.4 Output Functions 220
5.8.5 Output Control Variables 225
5.8.6 Console Screen 229
5.8.7 Hardware Interface 238
5.9 File System 249
5.9.1 Input Files 250
5.9.2 Output Files 255
5.9.3 File and Directory 262
5.9.4 Memory Image Files 266
5.10 User Interface 269
5.10.1 Top-level Loop 269
5.10.2 Read-eval-print Loop 271
5.10.3 Interrupt and Error Handling 273
5.10.4 User Querying 278
5.10.5 Line Editor 279

Chapter 6. Source File Library

6.1 Arrays 281
6.2 Irrational Functions 286
6.3 Structures 294
6.4 Flavors 297
6.5 Closures 304
6.6 Turtle Graphics 309
6.7 Mouse Interface 313

Chapter 7. Program Development System

7.1 Building and Saving the PDS 318
7.2 The Editor 319
7.2.1 The Editor Screen 320
7.2.2 Selecting Menu Options 321
7.2.3 Command Mode Menu Options 322
7.2.4 Edit Mode Commands 326
7.2.5 Theory of Operation 332
7.3 The Debugger 335
7.3.1 Command Mode Menu Options 335
7.3.2 Debugger Display 337
7.4 The Window Manager 339
7.4.1 Window Manager Commands 340
7.4.2 Window Manager Utilities 341

Chapter 8. Compiler

8.1 User Interface 346
8.2 Compilation Procedure 349
8.3 Performance Considerations 351
8.3.1 Interpreter Overhead 351
8.3.2 In-line Machine Code 352
8.3.3 Speed Improvements 352
8.4 Compiler Errors 354
8.4.1 Error Messages 354
8.4.2 Internal Errors 355
8.5 Potential Problems 356
8.5.1 Compiled Function Behavior 356
8.5.2 Name Conflicts 357
8.5.3 Redefining Primitives 358

Chapter 9. Machine-coded Routines

9.1 Data Representation 360
9.2 Argument Passing 362
9.3 Calling Machine-coded Routines 363
9.4 Loading and Linking 364

Chapter 10. LISP References

10.1 Professional Societies 367
10.2 Periodicals 368
10.3 Books 369

Appendixes

Appendix A. PC Video Modes 371
Appendix B. PC Screen Colors 372
Appendix C. PC Screen Characters 374
Appendix D. PC Extended Function Keys 376
Appendix E. Line Editor Commands 377
Appendix F. WordStar-like Editor Commands 378
Appendix G. EMACS-like Editor Commands 380
Appendix H. Base Page 382
Appendix I. Global Storage Area 389
Appendix J. Assembly Language Routine 390

Indexes

Function Index 391
Concept Index 409

Problem Report Form

License Agreement and Limited Warranty

Chapter 1. Introduction

Congratulations on your purchase of the **muLISP™ LISP Language Programming Environment**. It is a powerful system for implementing and delivering sophisticated software products on widely available MS-DOS based personal computers. The information in this manual is arranged as follows:

- **Chapter 1. Introduction** describes the major features of **muLISP**, the files on the master diskettes, the computer system requirements, how to make a working copy, and how the manual denotes keys on the keyboard.
- **Chapter 2. Fundamentals** describes how to start **muLISP**, how to enter commands, how to define functions, how to load source files, and how to load and save memory image files.
- **Chapter 3. Learning muLISP** describes how to load and play the demonstration game programs, and how to start the **muLISP** Tutorial System.
- **Chapter 4. Interpreter** describes the primitive data objects, the memory management system, the break options, how to interrupt a program, and the error messages.
- **Chapter 5. Primitives** describes the functions, special forms, macros, and control variables defined in the machine language kernel and in the COMMON.LSP source file.
- **Chapter 6. Source File Library** describes the facilities provided by the other source files included with **muLISP**. These facilities include arrays, irrational functions, structures, flavors, closures, turtle graphics, and mouse support.
- **Chapter 7. Program Development System** describes how to use the **muLISP** editor, debugger, and window manager.
- **Chapter 8. Compiler** describes how to compile functions using the **muLISP** native-code compiler.
- **Chapter 9. Machine Language Routines** describes how to write, load, and link machine language routines.
- **Chapter 10. LISP References** lists the societies, periodicals, and books relevant to LISP programming.

1.1 Major Features

muLISP consists of a machine language kernel, a program development system, a source file library, several demonstration game programs, a tutorial system, and this reference manual.

1.1.1 Machine Language Kernel

The machine language kernel is distributed as the executable COM file MULISP.COM. It includes the **muLISP** interpreter, garbage collector, and over 400 functions, special forms, and control variables. They are defined in machine language for maximum efficiency. Since the kernel consumes less than 50K (kilobytes) of memory, it leaves a large amount of computer memory free for application programs.

High performance has been achieved through the use of shallow variable binding, address typed data structures, and a closed pointer universe. The following summarizes the capabilities provided by the machine language kernel:

List Processing The kernel provides a large number of functions for selecting, constructing, recognizing, comparing, counting, searching, substituting, and modifying the elements of lists.

Merging and Sorting The kernel provides functions for efficiently merging and sorting lists using user-supplied predicates.

Arithmetic Operators The kernel provides operators (functions) for exact integer and rational arithmetic *and* adjustable precision approximate rational arithmetic. The radix used to express numbers is also adjustable.

Function Mapping The kernel provides functions and predicates for iteratively testing and/or manipulating the elements of any number of lists.

Character and String Processing The kernel provides functions for substring extraction, string searches, case conversion, and string comparisons (both case sensitive and insensitive). Hashing algorithm is used to rapidly intern new print names.

Communication Facilities The kernel provides functions for reading and writing character or binary data from or to the console, the system printer, or DOS files. Both sequential and random file access are supported.

Macro Characters The kernel's LISP reader can be modified by user-defined macro characters (terminating, non-terminating, and comment). For instance, the backquote character greatly simplifies macro definitions.

File System The kernel provides functions for opening, closing, renaming, and deleting files. Also provided are functions for changing the default DOS drive and path, determining a file date, and listing the files in a directory.

Memory Image Files The kernel provides functions for quickly loading and saving compact memory image files.

Control Constructs The kernel provides the following control constructs: "implicit PROGN", "implicit COND", COND, IF, PROGN, PROG1, LOOP, CATCH, THROW, UNWIND-PROTECT, and RETURN. The COMMON.LSP library file provides the following macro-defined control constructs: PROG2, WHEN, UNLESS, CASE, LET, LET*, DO, DO*, DOLIST, and DOTIMES.

Macros The **muLISP** interpreter allows macros to be expanded at run time or pseudo-compile time. Macros can be used to create user-defined control constructs. Functions can be defined to accept any number of arguments and/or to *not* evaluate their arguments.

Pseudo-code Compiler When a function is defined, the definition is automatically compiled into a compact pseudo-code. When a function definition is accessed, the definition is automatically decompiled. The use of pseudo-code to store function definitions reduces memory requirements by a factor of 3 *and* improves the speed of the interpreter.

Machine Language Routines If necessary for some special application or performance need, user-defined machine language routines can loaded and linked to the **muLISP** kernel.

Memory Management The kernel's two-pass, compacting garbage collector automatically reclaims unreferenced data structures. When necessary, data space boundaries are also adjusted to provide more room for heavily used spaces.

Subprocess Invocation The kernel provides functions for temporarily suspending **muLISP** in order to load and run an MS-DOS program or the MS-DOS command processor.

Generic MS-DOS The kernel will run on any of the wide variety of personal computers that use an MS-DOS compatible operating system. If running on a PC compatible computer, the kernel also supports color graphics in a variety of resolutions.

1.1.2 Program Development System

The **muLISP** program development system provides an interactive, window-based environment for software development.

Window Manager The window manager provides a uniform, menu-driven interface for application programs. It includes facilities for opening, splitting, closing, and changing windows.

Editor The screen-oriented text editor has been enhanced to facilitate the editing of LISP source files by incorporating such features as expression evaluation, blinking

parentheses, and undelete. With a single key stroke, editing can be suspended while you switch to a debug window or another edit window.

Debugger The debugger displays function trace output continuously or in single-step mode, making it easy to examine and/or modify intermediate results. Trace output is sent to a window separate from a program's output window.

Compiler The native code compiler *incrementally* compiles function definitions into native machine code, thereby improving the performance of speed-critical functions.

Runtime System After loading or compiling a **muLISP** application program, the program together with the **muLISP** Runtime System can be saved as a directly executable MS-DOS COM file or EXE file. This file can be made commercially available by agreeing to the terms of the **muLISP** Runtime License Agreement and paying the required fee. For details, write **Soft Warehouse, Inc.**, 3660 Waialae Avenue, Suite 304, Honolulu, Hawaii, 96816-3236, U.S.A.

1.1.3 Source File Library

The facilities provided by the source file library complement those provided by the machine language kernel.

Common LISP COMMON.LSP, along with the other source files, provides over 450 Common LISP functions, special forms, macros, and control variables.

Arrays ARRAY.LSP provides a set of functions and macros for creating, accessing, and recognizing multi-dimensional arrays.

Irrational Functions IRRATNAL.LSP provides functions for approximating the exponential, logarithmic, trigonometric, and inverse trigonometric functions to any desired degree of accuracy.

Structures STRUCTUR.LSP provides the DEFSTRUCT macro for creating named and unnamed record structures. Defining a structure in effect creates a new data type complete with accessors, constructors, recognizers, and copiers.

Flavors FLAVORS.LSP provides a set of functions and macros that facilitate object-oriented programming. It supports multiple inherited methods and instance variables, before and after demons, and whopper methods.

Closures CLOSURE.LSP provides functions for creating and manipulating dynamic closures of functions over a list of variables whose values are local to the closures.

Graphics GRAPHICS.LSP provides functions for doing point, line, and turtle graphics. Included as examples are functions for recursively drawing the popular "C" and "Dragon" curves.

Mouse Interface MOUSE.LSP provides functions for reading the position and button status of the Microsoft Mouse™.

1.1.4 Demonstration Game Programs

muLISP includes several interesting game and puzzle programs that demonstrate good **muLISP** and AI programming techniques.

Animal By asking a series of yes-or-no questions, this word game builds and uses a discrimination net to logically determine what animal the player has chosen.

Doctor This game, sometimes called **Eliza**, turns your computer into a psychologist of the Rogerian school of psychology, in which the doctor lets the patients resolve their own problems by asking leading questions.

Eights This program uses heuristic search techniques to "slide" eight "tiles" that are randomly arranged in a 3 by 3 "tray" into a standard configuration.

Tower of Hanoi This program uses character graphics and sound effects to display the recursive solution to this famous puzzle.

Metamind This program can play either side of this code breaking game. The object is to determine the colors and arrangement of four objects based on clues provided by the code maker.

1.1.5 Tutorial System and Documentation

Tutorial System The interactive tutorial system provides an on-line introduction to LISP programming. Topics include the basic LISP data structures and primitives, recursion and iteration, numerical programming techniques, and turtle graphics.

Help Facility The on-line help facility provides ready information on the arguments and/or initial value of the **muLISP** primitives, and the page number in the manual containing more detailed information.

muLISP Reference Manual This detailed manual describes all aspects of **muLISP** programming.

1.2 muLISP Diskette Files

The following files are on the master **muLISP** diskette(s):

- README — Information not yet included in this manual (display contents using the MS-DOS TYPE command)

Executable Files:

- MULISP.COM — **muLISP** interpreter (see Section 2.1)
- COMTOEXE.COM — COM file to EXE file conversion utility (see Section 5.9.4)
- HERCULES.COM — Resident program for Hercules-compatible graphics cards (see Section 5.8.6)

Program Development System:

- EDIT.LSP — Source file editor (see Section 7.1)
- DEBUG.LSP — Program debugger (see Section 7.2)
- WINDOWS.LSP — Window manager (see Section 7.3)
- ADDRESS.LSP — Sample data entry program (see Section 7.3)
- HELP.LSP — Help facility (see Section 3.1)

Source File Library:

- COMMON.LSP — Common LISP library (see Chapter 5)
- ARRAY.LSP — Array library (see Section 6.1)
- IRRATNAL.LSP — Irrational function library (see Section 6.2)
- STRUCTUR.LSP — Structure library (see Section 6.3)
- FLAVORS.LSP — Flavors library (see Section 6.4)
- CLOSURE.LSP — Dynamic closure library (see Section 6.5)
- GRAPHICS.LSP — Turtle graphics library (see Section 6.6)
- MOUSE.LSP — Microsoft Mouse library (see Section 6.7)
- NECGRAPH.LSP — NEC PC-9801 graphics library (see file)
- STATS.LSP — Memory usage statistics library (see file)

Demonstration Game Programs (see Section 3.1):

- ANIMAL.LSP Animal game
- ANIMAL.MEM Animal game's prebuilt data base
- DOCTOR.LSP Doctor or Eliza
- EIGHTS.LSP Eights puzzle
- HANOI.LSP Tower of Hanoi puzzle
- METAMIND.LSP Metamind game

Tutorial System (see Section 3.2):

- LESSONS.LSP Tutorial System lesson driver
- MULISP1.LES Data objects and primitive functions
- MULISP2.LES Defining functions using recursion
- MULISP3.LES Symbols, numbers, and conses
- MULISP4.LES List processing & iterative functions
- MULISP5.LES Numerical programming techniques
- MULISP6.LES Implementing turtle graphics functions

Native code Compiler (see Chapter 8):

- COMPILE.LSP Native code compiler loader
- COMPILE0.LSP User-interface and utility routines
- COMPILE1.LSP Function body code generator
- COMPILE2.LSP In-line code generator for special functions and forms
- COMPILE3.LSP Assembler and native code loader

1.3 System Requirements

muLISP runs on various Intel 8086, 8088, 80286, and 80386 based personal computers running the MS-DOS or compatible disk operating system. This section lists the minimum computer hardware and software required to run **muLISP**.

1.3.1 IBM PC Compatible Computers

- An IBM Personal Computer or Personal System/2 Computer running PC-DOS or the PS/2 operating system; *or* a PC compatible computer running Microsoft Corporation's MS-DOS operating system software, version 2.1 or later.
- A monochrome display adapter (MDA), a color graphics adapter (CGA), an enhanced graphics adapter (EGA), a video graphics adapter (VGA), a Hercules graphics adapter, *or* a compatible adapter installed in the computer.
- A monochrome monitor, a color monitor, an enhanced color monitor, *or* a multisync color monitor.
- At least 256K (kilobytes) of computer memory. **muLISP** can take advantage of up to 640K of memory.
- At least one double-sided (360K) 5 1/4 inch diskette drive *or* one double-sided (720K) 3 1/2 inch diskette drive.

1.3.2 NEC PC-9801 Compatible Computers

- An NEC PC-9801 Personal Computer *or* a PC-9801 compatible computer running Microsoft Corporation's MS-DOS operating system software, version 2.1 or later.
- A monochrome monitor *or* a color monitor.
- At least 256K of computer memory. **muLISP** can take advantage of up to 640K of memory.
- At least one double-sided (1.2M) 5 1/4 inch diskette drive *or* one double-sided (1.2M) 3 1/2 inch diskette drive.

1.3.3 Other MS-DOS Computers

- An Intel 8086, 8088, 80186, 80286, or 80386 based personal computer running Microsoft's MS-DOS operating system software, version 2.1 or later.
- At least 256K of computer memory. **muLISP** can take advantage of up to 640K of memory.
- At least one double-sided (360K) 5 1/4 inch diskette drive *or* one double-sided (720K) 3 1/2 inch diskette drive.

1.4 Making A Working Copy

As a convenience to authorized users, the **muLISP** diskette is normally *not* copy protected. It can be copied using standard DOS copying commands. In return, we ask that you abide by our copyright and *not* distribute copies of the software to unauthorized users.

As a safeguard against the loss, damage, or erasure of the **muLISP** diskette, you should make a working copy of the diskette *before* using **muLISP**. If the working copy is made on a diskette, the diskette must have a label bearing the following copyright notice:

muLISP™
Copyright © 1983, 1987, 1990 by Soft Warehouse, Inc.
All rights reserved worldwide

In addition to the copyright notice, it is a good idea to put the version number (i.e. 6.*nn*), the version date (i.e. *mm/dd/yy*), and the serial number on the diskette label. The version number and serial number are on the master **muLISP** diskette. The version number and version date are displayed when **muLISP** starts. When reporting problems, be sure to include the version number, version date, and serial number on the **muLISP** program **Problem Report Form** printed at the end of this manual.

Before making a working copy, we recommend write protecting the **muLISP** diskette. A 5 1/4 inch diskette is write protected by covering the notch on the right side of the diskette with a write-protect tab. A 3 1/2 inch diskette is write protected by moving the write-protect slide so the window is open. See your computer manuals for details.

1.4.1 Hard Disk Procedure

If your computer has a 5 1/4 inch or a 3 1/2 inch diskette drive for drive A and a hard disk drive for drive C, you can make a working copy of **muLISP** on the hard disk as follows:

- Turn on and boot the computer in the ordinary way from either the hard disk or a DOS system diskette in drive A.
- Insert the **muLISP** diskette in drive A.
- Switch to drive C by entering the DOS command

```
C:
```

- Make a new subdirectory named MULISP by entering the DOS command

```
MD MULISP
```

- Change to the MULISP subdirectory by entering the DOS command

```
CD MULISP
```

- Copy the files from the **muLISP** diskette to the MULISP subdirectory by entering the DOS command

```
COPY A:*.* C:
```

- After all the files have been transferred to the hard disk and the DOS prompt is redisplayed, remove the **muLISP** diskette from drive A and store it in a safe, cool, and dry place that is physically separated from the computer.

1.4.2 Diskette Procedure

If your computer has one or two diskette drives but no hard disk drive, you can make a working copy of **muLISP** on a diskette using the DISKCOPY.COM program described in the DOS manual.

Alternatively, if your computer has diskette drives for drives A and B, you can make a working copy of **muLISP** as follows:

- Turn on and boot the computer in the ordinary way from a DOS system diskette in drive A.
- Insert a blank diskette in drive B.
- Format the blank diskette by entering the DOS command

```
FORMAT B:
```

- Insert the **muLISP** diskette in drive A.
- Copy the files from the **muLISP** diskette to the blank working diskette by entering the DOS command

```
COPY A:*.* B:
```

- After all the files have been transferred to the working diskette and the DOS prompt is redisplayed, remove the **muLISP** diskette from drive A and store it in a safe, cool, and dry place that is physically separated from the working copy of **muLISP**.
- Remove the working diskette from drive B and place the previously described copyright notice on it.

1.5 Keyboard Notation

Throughout this manual keys on the computer keyboard are denoted by names printed in **bold** letters. Some keys may be marked with descriptive symbols instead of names that are spelled out. The following is an alphabetical list of the names and the corresponding symbols used on many keyboards:

Manual Name	Keyboard Symbol	Manual Name	Keyboard Symbol
Alt	Alt	**Ins**	Ins
Backspace	←	**Left arrow key**	←
Ctrl	Ctrl	**PgDn**	PgDn
Del	Del	**PgUp**	PgUp
Down arrow	↓	**Right arrow**	→
End	End	**Shift**	⇧
Enter	↵	**Space**	<space bar>
Esc	Esc	**Tab**	→|
Home	Home	**Up arrow**	↑

Sometimes you need to press two keys at the same time to obtain the desired effect. For example, when the manual says **Shift-Tab**, hold down the **Shift** key and press the **Tab** key. Similarly, **Ctrl-H** means hold down the **Ctrl** key and press the letter **H**, whereas **Alt-E** means hold down the **Alt** key and press the letter **E**.

When the manual says "enter a command", it means to type the command and then press the **Enter** key.

Chapter 2. Fundamentals

This chapter describes how to start **muLISP**, how to enter and edit commands, how to define functions, how to load source files, and how to load and save memory image files.

2.1 Starting muLISP

Once a working copy of the **muLISP** diskette has been made (see Section 1.4), it is a simple matter to start **muLISP** as follows:

- Turn on and boot your computer in the ordinary way from either a hard disk or a DOS system diskette in drive A. If you are not sure how to boot your computer, study the manuals that came with the computer and/or get help from someone experienced in using the computer.
- If your working copy of **muLISP** is on a diskette, insert the diskette in drive A and enter the command

```
A:MULISP
```

- If your working copy of **muLISP** is on a hard disk, enter the command

```
MULISP
```

  If this does not start **muLISP**, you will have to change to the drive and directory containing the file MULISP.COM before executing the above command. For example, the command

```
CD MULISP
```

  changes to the MULISP subdirectory if one exists. See your computer's DOS manual for details on how to change the drive and directory. Once you have changed to the correct drive and directory, you must reissue the MULISP command to actually start **muLISP**. If you are unable to start **muLISP** from the hard disk, you can always start it from a diskette as described earlier.

Whichever method you use to start **muLISP**, there is a short delay while the program loads from the diskette or hard disk. When complete, **muLISP** displays the header message

```
muLISP computer MS-DOS Version 7.nn (mm/dd/yy)
Copyright (c) 1983, 1987, 1990 Soft Warehouse, Inc.
All Rights Reserved Worldwide
```

Appropriate numbers will appear for the version number (7.*nn*), month (*mm*), day (*dd*), and year (*yy*). The version number and version date, along with the serial number, should be included in all inquiries concerning **muLISP**.

The *computer* in the header message indicates the type of computer system on which **muLISP** is being run. The computer type affects which console screen functions are operative (see Section 5.8.6).

The **muLISP** prompt is displayed below the header message. The default prompt consists of the $ character followed by a space and the console cursor. The prompt indicates the system is ready to accept console input. The prompt can be changed by modifying the string in the **muLISP** base page (see Appendix H).

2.2 Read-eval-print Loop

Following the prompt, you can enter LISP expressions written in list and/or dot notation. Multi-line expressions can be entered since an expression is not evaluated until all parentheses are balanced.

After you enter a complete LISP expression and press **Enter**, **muLISP** reads the expression, evaluates it, and prints the result beginning on a new line. This read-eval-print loop is repeated indefinitely until you enter a SYSTEM command, which terminates **muLISP** and returns control to DOS.

The following printout of a sample **muLISP** session demonstrates the read-eval-print loop. Comments are printed at the right margin preceded by semicolons, which is the **muLISP** comment character. Note that the comments need not be entered for the examples to work.

```
$ 'DOG                                              ;Quote a symbol
DOG

$ (+ 5 -2 4)                                        ;Add three integers
7

$ (EQUAL 'DOG 'CAT)                                 ;Compare two symbols
NIL

$ (member 'dog '(cat cow dog pig))                   ;Search a list
(DOG PIG)

$ (SYSTEM)                                          ;Quit muLISP, return to DOS

A>                                                  ;The DOS prompt
```

By default **muLISP** converts lower case letters to upper case as they are read. However, if the control variable *READ-UPCASE* is set to NIL, lower case letters are not converted to upper case (see Section 5.8.3).

As a programming convenience, the last three expressions read by the read-eval-print loop are assigned to variables: the last expression to +, the second from last to ++, and the third from last to +++. Also the evaluated result of the last three expressions read

are assigned to variables: the last result to *, the second from last to **, and the third from last to ***. For example:

```
$ (APPEND '(TOM SAM JOAN) '(ANN JOE SUE))
(TOM SAM JOAN ANN JOE SUE)

$ *
(APPEND '(TOM SAM JOAN) '(ANN JOE SUE))

$ **
(TOM SAM JOAN ANN JOE SUE)
```

The pre-defined read-eval-print loop may well be inappropriate for your application. You can define your own driver function by setting the control variable DRIVER to the name of your driver function (see Section 5.10.1).

muLISP output to the console can be "frozen" and then restarted using **Ctrl-S** as a toggle. This is useful to prevent the scrolling of text off the screen before it can be read.

2.3 Line Editing

Often while using **muLISP** you will need to enter and/or edit a line of text. A blinking cursor appears while you are typing in and/or editing the text. When you are finished, press **Enter** or **Ctrl-M** to actually enter the line.

Control keys are used for line editing. There is a control key associated with each letter of the alphabet. To enter a control key, press its associated letter while holding down the **Ctrl** key. On PC compatible computers, the arrow keys located at the right end of the keyboard can also be used for line editing (see Section 1.5).

To correct a mistake made while entering a line of text, press **Backspace**, **Rub**, or **Ctrl-H** to delete the mistake and then type the correction.

To make a change near the beginning of the line without deleting the characters at the end, press **Left arrow** or **Ctrl-S** to move the cursor to the left without erasing characters. Note that **Backspace** deletes characters as it moves the cursor left, whereas **Left arrow** does not. If you move the cursor too far to the left, press **Right arrow** or **Ctrl-D** to move the cursor to the right.

A **token** is a string of letters or numeric characters terminated by a space, a non-alphanumeric character, or the end of the line. Press **Ctrl-Left arrow** or **Ctrl-A** to move the cursor left a token. Press **Ctrl-Right arrow** or **Ctrl-F** to move it right a token.

Press **Home**, **Ctrl-]**, or **Ctrl-Q S** (i.e. press **Ctrl-Q** and then press **S**) to move to the left end of the line. Press **End**, **Ctrl-**, or **Ctrl-Q D** to move to the right end of the line.

If the cursor is within a sequence of characters, press **Del** or **Ctrl-G** to delete the character at the cursor. Press **Ctrl-T** to delete the token at and to the right of the cursor.

If you have so muddled the line you are editing that you would like a fresh start, press **Ctrl-Y** to delete the whole line. Press **Ctrl-Q Y** to delete the right end of the line beginning with the character at the cursor. Press **Ctrl-Q Backspace**, **Ctrl-Q Rub**, or **Ctrl-Q H** to delete the left end of the line.

The effect of typing characters when the cursor is positioned within a line depends on the current text entry mode. In **insert mode**, characters under and to the right of the cursor are moved right to make room for a new character. In **overwrite mode**, the character under the cursor is replaced by a new character. Press **Ins** or **Ctrl-V** to toggle between insert mode and overwrite mode.

Press **Tab** or **Ctrl-I** to move right to the next tab stop. Note that in insert mode characters to the right of the cursor are moved right by a tab command.

To insert a control character in the line of text, press **Ctrl-P** and then press the control key associated with the character. This prevents the control character from being interpreted as an editing command.

Press the **F3** function key or **Ctrl-U** to insert the text from the previous edit line into the current edit line, so it can be edited and then entered again.

These line editing commands are summarized in Appendix E. Note that they are deliberately similar to the editing commands provided by the full-screen, source file editor (see Section 7.1).

The line editing commands can be reassigned to different control keys by modifying the **muLISP** base page (see Appendix H).

2.4 Programming

Although it does not take full advantage of the capabilities of **muLISP**, the following is a valid recursive definition of a function for computing the factorial of a number:

```
(DEFUN FACTORIAL (N)
  (COND
    ((ZEROP N)  1)
    (T (* N (FACTORIAL (SUB1 N)))) ) )
```

The **muLISP** interpreter incorporates numerous extensions to LISP that significantly increase program readability and execution speed, while substantially decreasing the storage requirements for function definitions. The following are the significant extensions:

- When it creates a new symbol for a print name, **muLISP** sets the value of the new symbol to be itself. This auto-quoting of symbols eliminates the need to explicitly quote symbols that are not otherwise assigned a value.
- As with many LISPs, **muLISP** generalizes the COND special form so all the forms following a nonNIL predicate are evaluated in turn. The value returned by this extended COND is the value of the last form evaluated.
- As with many LISPs, **muLISP** uses an implicit PROGN when evaluating function definitions. This eliminates the need for an explicit PROGN to evaluate multiple forms. In addition, **muLISP** uses an implicit COND when evaluating definitions. This often eliminates the need for an explicit COND to implement a branch in a definition.

Comparing the following recursive definition of FACTORIAL with the previous one illustrates some of the extended capabilities of **muLISP**:

```
(DEFUN FACTORIAL (N)
  ((ZEROP N)  1)
  (* N (FACTORIAL (SUB1 N))) )
```

With the exception of the PROG and GO control constructs, LISP is an exemplary structured language. The following features of **muLISP** eliminates the need for these constructs:

- The multiple-exit LOOP control construct provides a convenient way to program structured nonrecursive loops without the need for a GO construct.
- In **muLISP**, excess arguments in a function's formal argument list can be used as local variables within the function. Also, in-line LAMBDA bodies and the LET and LET* control constructs can be used to introduce local variables in functions (see Section 5.6.1) without the need for a PROG construct.

Therefore the PROG and GO control constructs are not primitively defined in **muLISP**. However if necessary for compatibility, the source file COMMON.LSP does provide an interpreter for them.

Instead of requiring numerous explicit control constructs, **muLISP** programs can be guided primarily by the tree structure of function definitions. Program control using implicit PROGNs and implicit CONDs results in function definitions whose structures and meanings are more transparent, require less memory to store, and are much more efficient (see Section 5.6).

As an example of how to replace explicit control constructs with implicit PROGNs and CONDs, consider the following iterative definitions of the FACTORIAL function:

```
(DEFUN FACTORIAL (N)                                    ;Conventional definition
  (PROG (M)
    (SETQ M 1)
  A (COND
      ((ZEROP N)
        (RETURN M) ) )
    (SETQ M (* M N))
    (SETQ N (SUB1 N))
    (GO A) ) )

(DEFUN FACTORIAL (N M)                                  ;muLISP definition
  (SETQ M 1)
  (LOOP
    ((ZEROP N)  M)
    (SETQ M (* M N))
    (SETQ N (SUB1 N)) ) )
```

The first is a conventional nonrecursive definition (It is not a workable **muLISP** function definition unless COMMON.LSP is loaded). The second is an equivalent **muLISP** definition that is also nonrecursive, but is clearly more structured and elegant. Note that the **muLISP** definition requires only 28 conses; the conventional one requires 37 conses. This is a significant space saving for a LISP system running in a limited address space.

After defining FACTORIAL, you can generate some really large factorials in a hurry. For example,

```
$ (FACTORIAL 100)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

2.5 Source Files

Source files are ASCII text files that contain function definitions, variable assignments, and other commands. Numerous source files are included on the **muLISP** diskette (see Section 1.2). If *filename* specifies the drive, directory, and name of a source file, the command

```
(LOAD 'filename)
```

reads and evaluates the expressions in *filename*. If *filename* is not found, LOAD returns NIL. If *filename* does not specify a file name extension, the extension LSP is used. If *filename* does not specify a drive and/or directory, the file is read from the default DOS drive and/or directory. For example, the command

```
(LOAD 'C:\\GAMES\\ANIMAL)
```

loads the source file ANIMAL.LSP from the C:\GAMES drive and directory. Note that two \ characters are required in *filename* where only one appears in DOS pathnames because \ is the single escape character in LISP (see Section 5.8.1).

An alternative way to load a source file is to include its name in the DOS command line used to start **muLISP**. **muLISP** will automatically load the source file immediately after MULISP.COM is loaded. For example, the DOS command

```
A:MULISP C:\GAMES\ANIMAL.LSP
```

loads the source file ANIMAL.LSP from the C:\GAMES drive and directory after MULISP.COM is loaded from drive A.

When an attempt is made to load a source file using the DOS command line and the file is not found, the warning message

```
File not found
```

is displayed after the **muLISP** header message.

When loading a source file using either of these methods, the file name extension LSP can be omitted. However, if there is also a memory image file with the same name, it is loaded instead of the source file (see Section 2.6).

LOAD commands in one source file can be used for chain-loading other source files.

2.6 Memory Image Files

muLISP memory image files are compact binary files that contain the environment present at the time they are saved. The environment consists of all the data structures, variable values, property values, and function definitions in **muLISP**. The command

(SAVE '*filename*)

saves the current **muLISP** environment as a memory image file on the drive and directory specified by *filename*. If *filename* does not specify a file name extension, the extension SYS is used. If *filename* does not specify a drive and/or directory, the file is saved on the default DOS drive and/or directory. For example, the command

```
(SAVE '\\PACIFIC\\WHALES)
```

saves the current environment as the memory image file WHALES.SYS in the \PACIFIC directory on the default drive. Note that two \ characters are required where only one appears in DOS pathnames because \ is the single escape character in LISP (see Section 5.8.1).

If *filename* specifies the drive, directory, and name of a memory image file, the command

(LOAD '*filename*)

restores the environment present at the time the memory image file was saved. Since control is immediately transferred to the just loaded environment, LOAD does not return a value. However, if *filename* is not found, LOAD returns NIL. If *filename* does not specify a file name extension, the extension SYS is used. If *filename* does not specify a drive and/or directory, the file is read from the default DOS drive and/or directory. For example, the command

```
(LOAD '\\PACIFIC\\WHALES)
```

loads the memory image file WHALES.SYS from the \PACIFIC directory on the default drive.

The environment present at the time the LOAD command is issued is replaced with that stored in the memory image file. Thus unlike source files, loading memory image files is not cumulative. Instead the new file's environment destroys the existing environment, including all function definitions.

An alternative way to load a memory image file is to include the name of the file in the DOS command line used to start **muLISP**. The memory image file will automatically be loaded immediately after MULISP.COM is loaded. For example, the DOS command

```
MULISP B:\PACIFIC\WHALES
```

loads the memory image file WHALES.SYS from the \PACIFIC directory on the default drive after MULISP.COM is loaded from the default drive.

When an attempt is made to load a memory image file using the DOS command line and the file is not found, the warning message

File not found

is displayed after the **muLISP** header message.

When a memory image file is loaded using either of the above methods, the environment is restored exactly as it was at the time of the save. After loading the file, the amount of memory allocated to each of the various data spaces is adjusted according to the computer's current memory size. This means that the current memory size does not necessarily have to be the same as when the file was created. However if there is insufficient memory to store the new environment, an

Insufficient Memory, Aborting

error occurs and **muLISP** is terminated.

Memory image files are useful for saving intermediate results of an interactive session, so the session can be continued at a future time or after a computer failure. They can also be used to save function definitions entered directly using the read-eval-print loop (see Section 2.2). However, this method of program development provides no way of easily editing existing definitions. Thus it is practical for developing only the most trivial of programs.

The preferred method of program development is to write and debug source files using the Program Development System (see Chapter 7). When the program is perfected, the source files can be saved as a memory image file that can be quickly loaded.

Chapter 3. Learning muLISP

This chapter describes how to load and play the demonstration game programs and how to start the **muLISP** Tutorial System.

3.1 Demonstration Programs

Several demonstration game programs are included with the **muLISP** system. These programs demonstrate numerous programming techniques such as how to manipulate data structures and how to provide a friendly user interface. Each program can be loaded by issuing a LOAD command following the **muLISP** prompt. For example, to load the Animal Game from the disk in drive B, issue the command

```
(LOAD 'B:ANIMAL)
```

- ANIMAL.LSP Think of an animal. The program will try to guess the animal by asking simple yes-or-no questions. The Animal Game is a good example of the use of a database set up as a binary **discrimination net**. If the program fails to guess the animal, it asks you how to distinguish the animal it guessed from the animal you were thinking of. This allows the program to extend its database to include your animal. The more you play, the "smarter" the program becomes!

  The Animal Game database can be saved as a text file for use in a later session. ANIMAL.MEM is a prebuilt database containing over 100 animals that you can load.

- DOCTOR.LSP This is the **muLISP** version of the Eliza (sometimes called Doctor) program written by Joseph Weizenbaum of MIT. It turns your computer into an amateurish, but persistent, psychologist. It mimics the Rogerian school of psychology where the doctor asks leading questions, letting patients resolve their own problems.

  The "P:" (patient) prompt indicates that the "doctor" is waiting for you to comment on your problem. Don't forget to terminate your sentence with a period, question mark, or exclamation point. The "D:" (doctor) herald precedes the doctor's responses to your statements. To exit the program, just type "Goodbye."

  Most of the program consists of a script that contains the keywords, patterns, and responses used by the doctor. The script can be elaborated upon or replaced with a script on an entirely different subject.

- EIGHTS.LSP The Eights Puzzle consists of eight square "tiles" numbered 1 through 8 in a square "tray" that can accommodate nine tiles. The program asks the user to place the tiles on the tray in a scrambled order. Using a heuristic search, the program finds and displays a sequence of moves that rearranges the tiles into a standard order. A move consists of sliding a tile vertically or horizontally into the adjacent empty space.

The standard order can only be reached from half the initial positions. Before beginning the search, the program uses a quick, direct test to determine if the standard order is reachable. The test is based on the number of permutations between the initial and final positions.

- HANOI.LSP This program graphically demonstrates the recursive solution to the Tower of Hanoi Puzzle. There are three pegs, one of which has a stack of rings of successively smaller diameter from bottom to top (i.e. the smallest on the top). The objective is to transfer all the rings to one of the other pegs so they will be in the same order. Only one ring may be moved at a time, and at no time may a larger ring lie on a smaller ring.

  The program provides options that allow you to adjust the number of rings, the speed of operation, the use of sound effects, and the use of color. In normal speed, the program simulates ring movement by showing intermediate positions of the rings and by using sound effects. When HANOI.LSP is loaded, WINDOWS.LSP must be on the default drive and directory.

- METAMIND.LSP Metamind is similar to the popular Master Mind™ board game by Invicta. One player, the "code maker", chooses a secret code consisting of a sequence of four (not necessarily different) colors. The other player, the "code breaker", tries to break the code in a minimal number of guesses. In response to the code breaker's guess as to the code, the code maker states the number of exact matches (i.e. of the correct color in the correct position), and the number of correct colors that are in the wrong position. The computer can be either the code maker or breaker.

- HELP.LSP This file provides a help facility for the functions, special forms, macros, and control variables defined in the machine language kernel or the source file library. After HELP.LSP is loaded, commands of the form

  (HELP *name*)

  or

  (? *name*)

  display *name*'s arguments, primitive type, and the page number containing the detailed description of name.

3.2 Tutorial System

The **muLISP** Tutorial System is an interactive, on-line system designed to teach the fundamentals of LISP programming. The source file LESSONS.LSP contains the **muLISP** program that presents the lessons.

Each of the lesson files, named MULISP*n*.LES, contain the actual text and examples for each lesson. The following is a summary of the information presented in each lesson:

- MULISP1.LES Data objects and primitive functions
- MULISP2.LES Defining functions using recursion
- MULISP3.LES Symbols, numbers, and conses
- MULISP4.LES List processing & iterative functions
- MULISP5.LES Numerical programming techniques
- MULISP6.LES Implementing turtle graphics functions

The first time through, the lessons should be taken in order.

To run the Tutorial System, switch to the drive and directory containing LESSONS.LSP and the lesson files. Then start **muLISP** as described in Section 2.1, and issue the command

```
(LOAD 'LESSONS)
```

following the **muLISP** prompt.

After the lesson program is loaded, it will display a menu listing the available lessons with a brief indication of their contents. To start with the first lesson, simply press the **Space** bar. The screen will clear and the lesson will begin. From this point on the lessons should be self explanatory.

Chapter 4. Interpreter

This chapter describes the primitive data objects, the memory management system, the break options, how to interrupt a program, and the error messages.

4.1 Data Objects

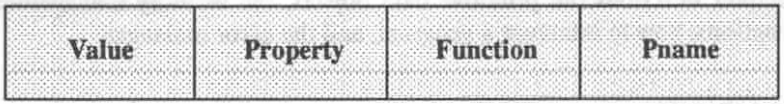
In **muLISP**, the primitively defined **data objects** consist of symbols, numbers, and conses. You can combine data objects to make **data structures** of arbitrary complexity. The type of a data object can be determined using the recognizer functions (see Section 5.1.1).

Each type of data object consists of a fixed number of **data cells**. Symbols and numbers consist of four data cells; conses consist of two data cells. Each data object has at least a **car cell** and a **cdr cell** (pronounced "could-er"). These two cells can only point to (i.e. contain the memory address of) data objects. Only the other cells of a data object are allowed to point to non-data objects such as character strings, number vectors, machine code, and pseudo-code.

The car and cdr cells form an interconnected network of data objects that is closed in the sense that you cannot leave the network by following the pointers. This closure eliminates the need for time consuming run-time type checking by the many functions that follow or manipulate these two data cells.

4.1.1 Symbols

A symbol is a data object consisting of four cells:

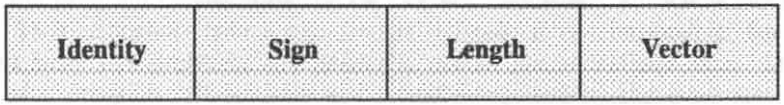


Symbols are uniquely stored in the sense that no two symbols can have identical print names. The four cells of a symbol are used as follows:

- **Value cell** This, the car cell, contains a pointer to the value of the symbol. The value cell can be changed by the value functions (see Section 5.3.1) and the modifier functions (see Section 5.2.3). The value cells of a function's formal arguments are also changed when the function is applied (see Section 5.7.2). When a symbol is created, its value cell is set to the new symbol itself. This automatic self-referencing of symbols is called **auto-quoting**.
- **Property list cell** This, the cdr cell, contains a pointer to the **property list** of the symbol. This list is used and modified by the property list functions (see Section 5.3.2). An atom (i.e. a symbol or a number) on the list is considered to be a flag; a cons is considered to be a property. A property consists of an indicator key consed to a property value. When a symbol is created, its property list cell is set to NIL (i.e. the empty list).
- **Function cell** This cell contains a pointer to the function definition of the symbol. Functions can be defined in either machine code or pseudo-code. When a function is applied, the interpreter uses the function cell to locate its definition. The definition primitives are used to change function definitions (see Section 5.3.3). When a symbol is created, its function cell is set to a routine that generates an Undefined Function error break when the symbol is applied as a function (see Section 5.10.3).
- **Pname cell** This cell contains a pointer to the symbol's **print name string**. The string consists of two bytes that store the length of the string followed by the ASCII characters used to print the symbol. Print name strings can be of arbitrary length. When a string is read by an input function or generated by a string function, a hashing algorithm is used to determine if a symbol already exists with that print name. If so, the existing symbol is used. If not, another symbol is created using the new print name. Once a symbol is created, its print name string cannot be modified.

4.1.2 Numbers

A number is a data object consisting of four cells:



Numbers can be further subdivided into **integers** and **ratios**. Internally, integers are further subdivided into **small integers** (those less than 65536 in magnitude) and **big integers**. Small integers are stored uniquely. Ratios and big integers are *not* stored uniquely, so numbers with the same numerical value can coexist in the system. Except for the function EQ, the distinction between small and big integers does not affect the **muLISP** programmer. The four cells of a number are used as follows:

- **Identity cell** This, the car cell, contains a pointer to the number itself. Thus numbers do not have to be quoted, since they evaluate to themselves. The content of this cell cannot be changed.
- **Sign cell** This, the cdr cell, determines the sign of the number and whether the number is a small integer, a big integer, or a ratio. The cell points to one of six symbols in accordance with the following table:

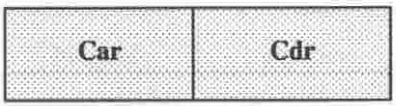
	Positive	Negative
Small integer	NIL	T
Big integer	LAMBDA	NLAMBDA
Ratio	MACRO	SPECIAL

  The content of the sign cell of a number can be determined by taking the cdr of the number, but the sign cell cannot be changed.
- **Length cell** If the number is a small integer, this cell contains the magnitude of the integer. If the number is a big integer, this cell contains the word length of the number's vector. If the number is a ratio, this cell contains a pointer to the numerator of the ratio, which must be an integer.
- **Vector cell** If the number is a small integer, this cell contains a hash link (i.e. a pointer to another small integer) or a 1. If the number is a big integer, this cell contains a pointer to the least significant two byte word of the integer's binary number vector. If the number is a ratio, this cell contains a pointer to the denominator of the ratio, which must be a positive integer.

  Number vectors are contiguous blocks of memory used to store the unsigned magnitude of big integers. The words of number vectors are kept in reverse order (i.e. more significant words are stored in successively lower memory addresses).

4.1.3 Conses

A cons is a data object consisting of two cells:



Binary trees are the most basic type of data structure in **muLISP**. Internally they are implemented as a network of cell pairs called **conses**. Each cons consists of a car cell and a cdr cell. As mentioned earlier, the cells of a cons can only point to *bona fide* data objects: either symbols, numbers, or conses.

Conses are sometimes called **dotted-pairs**, because of the **dot notation** sometimes used to represent conses. If obj_1 and obj_2 are data objects, the expression

$$(obj_1 \ . \ obj_2)$$

denotes a cons whose car cell points to obj_1 and whose cdr cell points to obj_2. Although cumbersome, dot notation is capable of showing the structure of any binary tree.

In most applications it is more convenient to think in terms of linear **lists** rather than deeply nested binary trees. Therefore cons structures are usually expressed in **list notation.** The list

$$(obj_1 \ obj_2 \ ... \ obj_n)$$

represents a chain of conses linked through their cdr cells and whose car cells point to the elements of the list. To be a valid list, the cdr cell of the last cons of the list must point to the atom NIL. The object above expressed in dot notation is

$$(obj_1 \ . \ (obj_2 \ . \ (... \ . \ (obj_n \ . \ \text{NIL})...)))$$

The function READ accepts both dot and list notation (see Section 5.8.1). In list notation, the **null list** or **empty list** is denoted by NIL. Therefore, when "( )" is read by READ, it returns NIL.

The output functions WRITE, PRIN1, PRINC, and PRINT use list notation to the maximum extent possible when printing data objects (see Section 5.8.4). For example,

```
(DOG . (CAT . COW))
```

is displayed by these functions as

```
(DOG CAT . COW)
```

4.2 Memory Management

Dynamic, automatic memory management gives LISP much of its inherent power. This frees the programmer from having to allocate, in advance, sufficient memory for a problem. Such allocation before run-time is difficult, if not impossible, for most AI applications since the problems are generally not of predetermined size. During the execution of a program, all memory not actually required by the program should be available for the creation of new data objects. The following subsections describe how **muLISP** achieves this.

4.2.1 Data Spaces

When **muLISP** starts, it partitions available memory into several distinct data spaces. Unfortunately the address space of computers based on microprocessors in the Intel 8086 family is divided into 64K (kilobytes) segments. This relatively small segment size arbitrarily limits the maximum size of each **muLISP** data space. The following summarizes the contents of each data space and gives maximum size in kilobytes:

- **Atom Space** This space stores symbols and numbers. The symbols and numbers are based at opposite ends of the space. Each atom requires four data cells (8 bytes). The maximum size of the space is 128K.
- **Cons Space** This space stores conses. Each cons requires two data cells (4 bytes). The maximum size of the space is 128K.
- **String Space** This space stores the print names of symbols as character strings. Each string requires two bytes for its length plus one byte for each character in the string. The maximum size of the space is 64K.
- **Vector Space** This space stores the magnitude of big integers as binary number vectors. Each vector requires as many two byte words as is necessary to store the magnitude of the number. The maximum size of the space is 64K.
- **Code Space** This stores user-defined function definitions as pseudo-code. There can be up to six code spaces (see NEW-CODE-SPACE in Section 5.7.6). The maximum size of each space is about 55K.
- **Stack Space** This space stores the control stack and the variable stack. The stacks are based at opposite ends of the space. The maximum size of the space is about 32K.

muLISP can take advantage of up to 640K of computer memory. However, it may happen that one of the data spaces becomes exhausted even though other spaces still have some free space. If this situation should occur during the execution of a program, a Memory Full error break occurs (see Section 4.5).

4.2.2 Garbage Collection

The nature of stack operations makes management of the stack space automatic and continuous. However, management of the remaining spaces requires explicit recycling of data objects that are no longer needed. New data objects are constantly being *created* during the execution of a program, while others are implicitly *discarded* when they are no longer referenced. When the creation process uses up the available resources in any one of the data spaces, a **garbage collection** is performed on *all* data spaces. Storage space vacated by discarded data objects is reclaimed so the program can continue.

A two-pass, mark-sweep garbage collection algorithm is used. The *first* pass consists of marking all active data objects. These are the objects accessible by means of chaining through data cells beginning from the value and property list cells of all system symbols, or from the variable stack, or from pseudo-code. Unreferenced, auto-quoted symbols that have no properties and have no current function definition are not marked. Unmarked symbols are removed during the second pass.

During the *second* pass of the garbage collection, all the marked data compaction objects are collected and compacted into one end of their respective data spaces. This leaves the remainder of each data space available for newly created objects.

4.2.3 Data Space Reallocation

After a garbage collection, one or more of the data spaces may have an insufficient amount of free storage to permit a program to continue execution, despite the fact that the other data spaces may have abundant free space. If this situation occurs, a data space reallocation routine is called to move the partitions dividing the spaces, giving more memory to the exhausted space or spaces. (However, the limits explained in Section 4.2.1 on the size of each data space must be observed by the reallocator.) Thus **muLISP** automatically responds to changing demands placed on the various data spaces by programs.

Although garbage collection and data space reallocation are automatic, their occurrence is not entirely invisible to the user since they cause a short pause in the execution of a program. On PC compatible computers, the blinking cursor temporarily changes to a full block during collections and reallocations.

The amount of time required for a collection or reallocation depends on the amount of data currently in the system and the clock speed of the computer. Garbage collections and data space reallocations typically require a second or two each. Normally this is of no concern to the user, but it should be considered in the design of real-time applications.

A phenomenon known as **thrashing** occurs when the system is forced to spend an inordinate amount of time garbage collecting for a very small return of data space. The symptom is greatly increased execution time for a given task. It can sometimes be resolved by increasing the computer's memory size (up to 640K) and/or modifying the program to reduce storage requirements.

4.3 Break Option Menu

When a console interrupt occurs (see Section 4.4) or a program error is detected (see Section 4.5), execution of the program is suspended and a message is displayed indicating the nature of the break. This is followed on the next line by the break option menu:

Continue, Break, Abort, Top-level, Restart, System

You select one of the options by entering the first letter of its name (C, B, A, T, R, or S). Note that the options are listed in order of increasingly drastic action.

- Continue This option returns control to the program that invoked the break routine. If the break was caused by a console interrupt, execution continues as if the break had not occurred. If the break was caused by an error trap, the form passed to the break by the error handler is returned as the value of the errant function.
- Break This option temporarily suspends program execution and calls the current driver function (see Section 5.10.2). The default driver function implements the read-eval-print loop (see Section 2.2). This allows you to examine and/or change variable bindings, function definitions, etc. To exit the break and resume program execution, press the **Esc** key following the dollar-sign prompt.
- Abort This option aborts program execution, unwinds the variable stack (i.e. restores formal arguments to their original value), and returns control to the current driver function (see Section 5.10.2). Function definitions, property values, and global variables are left unchanged.
- Top-level This option aborts program execution, unwinds the variable stack, makes the console the standard input source and output sink, and returns control to the top-level loop (see Section 5.10.1). Function definitions, property values, and global variable values are left unchanged.
- Restart This option closes all open files, abandons the environment, and initiates a fresh **muLISP** system. All variable bindings, user-defined functions, and property values in the current environment are destroyed.
- System This option closes all open files, terminates execution of **muLISP**, and returns control to the host operating system.

4.4 Console Interrupts

While debugging, you can initiate a console interrupt to stop a "runaway" or nonterminating program, and return control to the console. Press the **Esc** key during the execution of a program to initiate the interrupt. When the console interrupt character is detected, the message

Console Interrupt Break

is displayed on the console followed by the break option menu (see Section 4.3). For console interrupt breaks, the Abort option is normally the appropriate option to choose.

The character used to initiate console interrupts can be changed by modifying the Default Read Table in the **muLISP** base page (see Appendix H).

Programs can disable console interrupts by reassigning the control variable *INTERRUPT-HOOK* to NIL (see Section 5.10.3). Note that while debugging programs that disable console interrupts, the only way to interrupt them is to restart the computer.

4.5 Error Messages

When a program is running and an error condition is detected by **muLISP**, an appropriate error message is displayed on the console, followed by the form that provoked the error and the break option menu (see Section 4.3). The following is an alphabetical list of all the error messages and their significance (**muLISP** compiler errors are described in Chapter 8):

- Dimension Error occurs when MAKE-ARRAY is called with other than a positive integer or a list of positive integers for the dimensions of the array, or when it is called with initial contents incompatible in length with the dimensions of the array (see Section 6.1).

- Disk Full occurs when there is insufficient disk space to store the data being written to a file. When this occurs, a throw is made to the label Disk Full with a value of NIL (see Section 5.8.4). If not caught by the program, the top-level loop catches the throw and generates an error break (see Section 5.10.1).

  Since the file remains open for output, programs can allow the user to delete other files from a full diskette, and then continue writing to the file.

- End-Of-File occurs when an attempt is made to read from beyond the end of a file or from "holes" within a file (see FILE-WRITE-POSITION in Section 5.9.2). The full pathname of the file being read is displayed after the error message.

- Exact Mode occurs when exact mode arithmetic is in effect (i.e. the control variable *EXACT-MODE* is neither NIL nor the symbol RATIONAL) and an irrational numerical function (see Section 6.2) is called upon to generate an approximate result.

- File not found occurs when the source or memory image file given on the DOS command line used to start **muLISP** is not found *or* is a memory image file created under a different version of **muLISP** (see Section 2.6). Memory image files can only be loaded under the same version of **muLISP** used to create them.

  Note that when LOAD fails to find a source or memory image file, it returns NIL instead of displaying a warning message (see Section 5.9.4).

- Include Error occurs when DEFSTRUCT is called upon to include a named structure in an unnamed structure, or to include an unnamed structure in a named structure (see Section 6.3).

- Incompatible Definition occurs when DEFSTRUCT is called upon to create a new definition for a structure function that is incompatible with the existing definition (see Section 6.3).

- Insufficient Arguments occurs when a function is called without enough arguments.

- Insufficient Memory, Aborting occurs when there is not enough memory to load a memory image file. Since there is no other recourse, **muLISP** is immediately terminated and control returns to DOS (see Section 2.6).

  Note that a memory image *can* be loaded on a computer with less memory than the computer on which the memory image was created. This error occurs only when the computer on which the memory image is being loaded does not have enough memory to store the memory image. The only way to load the memory image file is to obtain more memory for your computer.

- Invalid Argument occurs when an irrational numerical function is called with an argument for which the function does not have a real value (see Section 6.2).

- Invalid Place Form occurs when SETF, INCF, or DECF is called with an invalid place form (see Section 5.3.1).

- Keyword Not Handled occurs when MAKE-INSTANCE is called with keywords that were not handled by the inherited and included flavors of the flavor (see Section 6.4).

- Memory Full occurs when there is insufficient memory to continue execution of a program. When this occurs, a throw is made to the label Memory Full with a value of NIL. If not caught by the program, the top-level loop catches the throw and generates an error break (see Section 5.10.1).

Normally, the memory management system provides sufficient storage in each data space to continuously satisfy the demands of programs (see Section 4.2). This error occurs if the memory requirements for storing data objects exhaust all available resources. Statistics in the following form are displayed before the error break message:

GC: *counter atom cons string vector code stack*

The hexadecimal numbers show the available space and the total space in each of the data spaces (see Section 5.7.6). You can use the statistics to help determine the space responsible for the memory exhaustion. However, memory exhaustion can occur even if the statistics show some memory available in all spaces (see Section 4.2.1).

- Missing Required Flavor occurs when MAKE-INSTANCE is called upon to create an instance of a flavor that did not include all the required flavors (see Section 6.4).

- Noninteger Argument occurs when LOGAND, LOGIOR, LOGXOR, LOGNOT, SHIFT, or INTEGER-LENGTH is called with a noninteger argument (see Section 5.4.5).

- Nonnumeric Argument occurs when a numeric function is called with a nonnumeric argument (see Section 5.4).

- Nonsymbolic Argument occurs when SET, SETQ, PSETQ, POP, PUSH, INCQ, DECQ, or DEFSTRUCT is called with a nonsymbolic argument.

- Syntax Error occurs when READ reads either an unmatched right parenthesis (i.e. too many right parentheses) or an error in dot notation, for example

```
(A . )
```

or

```
(A B . C D)
```

Since this error break is generated by the right parenthesis and comma character macros, redefining these character macros enables the error to be handled by a program (see Section 5.8.1).

- Unclaimed Message occurs when SEND is called upon to send a message to an instance of a flavor that does not have a method to handle the message (see Section 6.4).

- Undefined Function occurs when an attempt is made to apply a symbol having no function definition. Following the error message is the form that made the call on the undefined function. A common course of action when confronted with this error is to

select the Break option, appropriately define the undefined symbol, and then continue with the original program by issuing the command

```
(RETURN (EVAL BREAK))
```

- Undefined Structure occurs when DEFSTRUCT is called upon to include an undefined structure in a structure (see Section 6.3).
- Unnamed Structure occurs when DEFSTRUCT is called upon to define a predicate for an unnamed structure (see Section 6.3).
- Unsupported Option occurs when DEFSTRUCT is called with an option not supported by the structure facility (see Section 6.3).
- Zero Divide occurs when /, FLOOR, CEILING, TRUNCATE, ROUND, MOD, REM, or DIVIDE is called with a divisor of zero (see Section 5.4).

Chapter 5. Primitives

This chapter describes the functions, special forms, macros, and control variables defined in the machine language kernel and in the COMMON.LSP source file. All user-defined functions and macros must ultimately be defined in terms of these primitives.

The title line of each primitive indicates its name, its arguments or initial value, and its type. The primitive name is printed in Helvetica type at the left margin.

If the primitive is a function, special form, or macro, its name is followed by a list of names, enclosed in square brackets and separated by commas, that represent each of its arguments. The type of the primitive is printed at the right margin.

If the primitive is a control variable, its name is followed by a colon and its initial value. The phrase "Control variable" is printed at the right margin.

If the primitive is defined in COMMON.LSP, the primitive type printed at the right margin is preceded by the word "Library". These library primitives are only available after issuing a command of the form

(LOAD '*path*\\COMMON.LSP)

where *path* specifies the drive and directory containing the file COMMON.LSP.

Following the title line is a description of the primitive. If the primitive is a function, special form, or macro, its arguments, value, and side-effects, if any, are described. If the primitive is a control variable, its effect on the system is described. There may also be a discussion of subtle points and common uses of the primitive.

Following the plain language description, there is usually a formal definition of the primitive, written in **muLISP**. Functions are defined using DEFUN. Special forms and macros are defined using DEFMACRO.

Occasionally auxiliary functions are introduced to aid in formal definitions. Lower case letters are used for the names of auxiliary functions to indicate that they are not directly accessible to the user. When irreducible, primitive concepts are introduced in formal definitions, they are described in English using lower case letters.

Finally, if appropriate, an interaction is shown that illustrates the use of the primitive. If the expression following the dollar-sign ($) is entered by the user, **muLISP** will evaluate the expression and return the result exactly as shown on the next line of the example. Comments concerning an example are shown to the right of the example following a semicolon, which is the comment delimiter.

5.1 Predicates

A **predicate** is a function used to test if its arguments satisfy some condition. If the test is false, the predicate returns NIL. If the test is true, it returns a nonNIL value, usually T. Predicates are used in conjunction with conditional control constructs (see Section 5.6) to guide the interpreter through function definitions.

5.1.1 Recognizers

A **recognizer** is a predicate used to determine the type of a data object. If its argument is of the specified type, the recognizer returns T, otherwise it returns NIL.

SYMBOLP [*object*] Function
STRINGP [*object*] Library function

If *object* is a symbol, (SYMBOLP *object*) returns T, otherwise it returns NIL.

In **muLISP**, no distinction is made between strings and symbols. Thus the definition of STRINGP is *identical* to that of SYMBOLP and is included in COMMON.LSP solely for compatibility.

```
(DEFUN SYMBOLP (OBJ)
  ((if OBJ is a symbol)  T)
  NIL )

$ (SYMBOLP 'DOG)                                    ;DOG is a symbol
T
$ (SYMBOLP ())                                ;The empty list (= NIL) is a symbol
T
$ (SYMBOLP 100)                                    ;Integers are not symbols
NIL
$ (SYMBOLP 2/3)                                    ;Numbers are not symbols
NIL
$ (SYMBOLP 4.235)                                  ;Numbers are not symbols
NIL
$ (SYMBOLP '(A B C))                          ;Nonempty lists are not symbols
NIL
```

INTEGERP [*object*] Function
NUMBERP [*object*] Function
RATIONALP [*object*] Library function

If *object* is an integer, (INTEGERP *object*) returns T, otherwise it returns NIL.

If *object* is a number, either an integer or a rational, (NUMBERP *object*) returns T, otherwise it returns NIL.

In **muLISP**, rational numbers also serve as floating point numbers. Thus the definition of RATIONALP is *identical* to that of NUMBERP and is included in COMMON.LSP solely for compatibility.

```
(DEFUN INTEGERP (OBJ)
  ((if OBJ is an integer) T)
  NIL )

(DEFUN NUMBERP (OBJ)
  ((INTEGERP OBJ))
  ((if OBJ is a rational number) T)
  NIL )

$ (INTEGERP 100)                                    ;100 is an integer
T

$ (INTEGERP 2/3)                                  ;Ratios are not integers
NIL

$ (INTEGERP 4.235)                              ;Decimal numbers are not integers
NIL

$ (NUMBERP 100)                                     ;Integers are numbers
T

$ (NUMBERP 2/3)                                     ;Ratios are numbers
T

$ (NUMBERP 4.235)                               ;Decimal numbers are numbers
T
```

ATOM [*object*] Function

If *object* is an atom, either a symbol or a number, (ATOM *object*) returns T, otherwise it returns NIL.

```
(DEFUN ATOM (OBJ)
  (OR (SYMBOLP OBJ) (NUMBERP OBJ)) )

$ (ATOM 'DOG)                                          ;Symbols are atoms
T

$ (ATOM 100)                                           ;Integers are atoms
T

$ (ATOM 2/3)                                           ;Numbers are atoms
T

$ (ATOM 4.235)                                         ;Numbers are atoms
T

$ (ATOM '(A B C))                          ;Nonempty lists are not atoms
NIL

$ (ATOM '(A . B))                                      ;Conses are not atoms
NIL

$ (ATOM ())                                ;The empty list (= NIL) is an atom
T
```

CONSP [*object*] Function
LISTP [*object*] Function

If *object* is a cons, (CONSP *object*) returns T, otherwise it returns NIL.

If *object* is a list, either a cons or the atom NIL, (LISTP *object*) returns T, otherwise it returns NIL.

```
(DEFUN CONSP (OBJ)
  (NOT (ATOM OBJ)) )

(DEFUN LISTP (OBJ)
  (OR (CONSP OBJ) (NULL OBJ)) )

$ (CONSP 'DOG)                                    ;Symbols are not conses
NIL

$ (CONSP 4.235)                                   ;Numbers are not conses
NIL

$ (CONSP '(A B C))                                    ;Lists are conses
T

$ (CONSP ())                                ;The empty list (= NIL) is not a cons
NIL

$ (LISTP 'DOG)                                    ;Symbols are not lists
NIL

$ (LISTP 4.235)                                   ;Numbers are not lists
NIL

$ (LISTP '(A B C))                                    ;(A B C) is a list
T

$ (LISTP ())                                 ;The empty list (= NIL) is a list
T
```

NULL [*object*] Function
ENDP [*object*] Function

If *object* is the empty list, (NULL *object*) returns T, otherwise it returns NIL. The empty or null list is represented by the symbol NIL.

If *object* is the empty list, (ENDP *object*) returns T. If *object* is a cons, ENDP returns NIL. Otherwise (i.e. if *object* is an atom), ENDP generates a Non-list Object error break.

Since the atom NIL represents both the empty list and the truth value false, the functions NULL and NOT always return identical values. However for program readability, it is better to use NULL when testing for the empty list and to use NOT when testing for the truth value false. When debugging it is even better to use ENDP instead of NULL when testing for the empty list, since an error break occurs if ENDP is given a non-list.

```
(DEFUN NULL (OBJ)
  (EQ OBJ NIL) )

(DEFUN ENDP (OBJ)
  ((NULL OBJ))
  ((ATOM OBJ)
    (BREAK (LIST 'ENDP OBJ) "Non-list Object") ) )

$ (NULL NIL)                                          ;NIL is the empty list
T

$ (NULL ())                                           ;() is the empty list
T

$ (NULL 'DOG)                              ;A nonNIL symbol is not the empty list
NIL

$ (NULL '(A B C))                          ;A nonempty list is not the empty list
NIL

$ (NULL '(NIL))                            ;A nonempty list is not the empty list
NIL

$ (ENDP NIL)                                          ;NIL is the end of a list
T

$ (ENDP '(A B C))                                ;A nonempty list is not the end
NIL

$ (ENDP 'DOG)                                      ;A nonNIL symbol is not a list
Non-list Object Break: (ENDP DOG)
```

5.1.2 Comparators

A **comparator** is a predicate used to compare two data objects. If its arguments satisfy the comparison, the comparator returns T, otherwise it returns NIL.

EQ [$object_1$, $object_2$] Function
NEQ [$object_1$, $object_2$] Function

If $object_1$ is identical to $object_2$, (EQ $object_1$ $object_2$) returns T, otherwise it returns NIL.

(NEQ $object_1$ $object_2$) is equivalent to (NOT (EQ $object_1$ $object_2$)).

EQ tests for the *identity* of its two arguments (i.e. they point to the same data object in memory). Since symbols and small integers are stored uniquely (see Section 4.1), EQ is an efficient way to determine if two symbols or two small integers are equal.

```
(DEFUN EQ (OBJ1 OBJ2)
  (= (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (EQ 'Alice 'Alice)                              ;Identical symbols are EQ
T

$ (EQ 7 (+ 3 4))                                ;Equal small integers are EQ
T

$ (EQ 100000 100000)                          ;Equal big integers may not be EQ
NIL

$ (EQ 4.75 4.75)                            ;Equal decimal numbers may not be EQ
NIL

$ (EQ '(A B C) '(A B C))                            ;Equal lists may not be EQ
NIL
```

EQL [*object*$_1$, *object*$_2$] Function
NEQL [*object*$_1$, *object*$_2$] Function

If *object*$_1$ is identical to *object*$_2$ or if both are numbers having the same numeric value, (EQL *object*$_1$ *object*$_2$) returns T, otherwise it returns NIL.

(NEQL *object*$_1$ *object*$_2$) is equivalent to (NOT (EQL *object*$_1$ *object*$_2$)).

Since big integers and ratios are *not* stored uniquely (see Section 4.1), EQL rather than EQ should be used to determine if two numbers are equal. Nonatomic arguments are EQL if and only if they point to the same cons in memory.

```
(DEFUN EQL (OBJ1 OBJ2)
  ((AND (NUMBERP OBJ1) (NUMBERP OBJ2))
    (= OBJ1 OBJ2) )
  (= (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (EQL 'Alice 'Alice)                          ;Identical symbols are EQL
T

$ (EQL 7 (+ 3 4))                                 ;Equal numbers are EQL
T

$ (EQL 100000 100000)                             ;Equal numbers are EQL
T

$ (EQL 4.75 4.75)                                 ;Equal numbers are EQL
T

$ (EQL '(A B C) '(A B C))                      ;Equal lists may not be EQL
NIL
```

EQUAL [*object*$_1$, *object*$_2$, *test*] Function
TREE-EQUAL [*object*$_1$, *object*$_2$, *test*] Library function

If *object*$_1$ and *object*$_2$ are isomorphic tree structures with leaves such that (*test leaf*$_1$ *leaf*$_2$) is nonNIL, (EQUAL *object*$_1$ *object*$_2$ *test*) returns T, otherwise it returns NIL. If the *test* argument is NIL or not given, EQUAL uses EQL for *test*. The definition of TREE-EQUAL is *identical* to that of EQUAL and is included in COMMON.LSP solely for compatibility.

Thus, EQUAL tests for the *equality* of its two arguments. Two nonatomic arguments are EQUAL if the car branches are EQUAL and the cdr branches are EQUAL. Two EQUAL data structures look the same when printed. Since EQL is faster, it should be used in place of EQUAL in cases where at least one of the arguments is known to be atomic.

```
(DEFUN EQUAL (OBJ1 OBJ2 TEST)
  ((ATOM OBJ1)
    (FUNCALL (OR TEST 'EQL) OBJ1 OBJ2) )
  ((ATOM OBJ2)  NIL)
  ((EQUAL (CAR OBJ1) (CAR OBJ2) TEST)
    (EQUAL (CDR OBJ1) (CDR OBJ2) TEST) ) )

$ (EQUAL 'Alice 'Alice)                          ;Identical symbols are EQUAL
T

$ (EQUAL 7 (+ 3 4))                                 ;Equal numbers are EQUAL
T

$ (EQUAL 4.75 4.75)                                 ;Equal numbers are EQUAL
T

$ (EQUAL '(A B C) '(A B C))                          ;Equal lists are EQUAL
T

$ (EQUAL '(A B C) '(C B A))                       ;Nonequal lists are not EQUAL
NIL
```

TAILP [$list_1$, $list_2$] Function

If $list_1$ is a sublist (i.e. one of the top-level conses) of $list_2$, (TAILP $list_1$ $list_2$) returns T, otherwise it returns NIL.

```
(DEFUN TAILP (LST1 LST2)
  ((EQ LST1 LST2))
  ((ATOM LST2)  NIL)
  (TAILP LST1 (CDR LST2)) )

$ (SETQ FOO '(A B C D))                                    ;Initialize FOO
(A B C D)

$ (TAILP '(C D) FOO)                                       ;Not a tail of FOO
NIL

$ (SETQ BAZ (CDDR FOO))                                    ;Initialize BAZ
(C D)

$ (TAILP BAZ FOO)                                          ;BAZ is a tail of FOO
T
```

ORDERP [*object*$_1$, *object*$_2$] Function

If *object*$_1$ is ordered before *object*$_2$, (ORDERP *object*$_1$ *object*$_2$) returns T, otherwise it returns NIL.

ORDERP provides an efficient **chronological ordering** for atoms that is useful for writing sorting and searching routines. Numbers are ordered before symbols, which are ordered before conses. Two numbers are ordered in increasing numerical order. Two symbols are ordered by comparing their physical memory address (i.e. according to their order of introduction). Two conses are ordered by comparing their physical memory address.

```
(DEFUN ORDERP (OBJ1 OBJ2)
  ((NUMBERP OBJ1)
    ((NUMBERP OBJ2)
      (< OBJ1 OBJ2) )
    T )
  ((NUMBERP OBJ2)  NIL)
  (< (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (ORDERP 5 9)                                  ;Numbers are ordered numerically
T

$ (ORDERP 'DOG 5)                              ;Symbols are ordered after numbers
NIL

$ (ORDERP 'DOG 'CAT)                              ;DOG was introduced before CAT
T

$ (ORDERP 'CAT 'DOG)                           ;CAT was not introduced before DOG
NIL
```

5.1.3 Logical Operators

A **logical operator** is a predicate used to perform Boolean operations on the truth values of its arguments. NIL has the truth value false. All other data objects have the truth value true.

NOT [*object*] Function

If *object* is the atom NIL, (NOT *object*) returns T, otherwise it returns NIL.

Since the atom NIL represents both the empty list and the truth value false, the functions NULL and NOT always return identical values. However for program readability, it is better to use NULL when testing for the empty list and to use NOT when testing for the truth value false.

```
(DEFUN NOT (OBJ)
  (EQ OBJ NIL) )

$ (NOT NIL)                                    ;NIL has truth value false
T

$ (NOT '())                                     ;() has truth value false
T

$ (NOT 'FOO)                                ;NonNIL symbols have truth value true
NIL

$ (NOT '(A . B))                                ;Conses have truth value true
NIL

$ (NOT (EQ 'DOG 'CAT))                                ;DOG is not EQ to CAT
T
```

AND [$form_1$, $form_2$, ..., $form_n$] Special form

(AND $form_1$ $form_2$ ... $form_n$) evaluates each form in turn until one of them evaluates to NIL or all the forms are evaluated. If a form evaluates to NIL, AND returns NIL, otherwise it returns the value of $form_n$. The call (AND) returns T, an identity for the AND operator. (AND $form_1$ $form_2$ ... $form_n$) is equivalent to the call

(IF $form_1$ (IF $form_2$ (IF ... (IF $form_{n-1}$ $form_n$))))

Note that AND evaluates a form if and only if all the preceding forms have evaluated to nonNIL values.

```
(DEFMACRO AND LST
  ((NULL LST))
  ((NULL (CDR LST))
    (CAR LST) )
  (LIST 'IF (CAR LST) (MACROEXPAND (CONS 'AND (CDR LST)))) )

$ (AND (EQ 'DOG 'CAT) (< 2 3))                          ;One comparison is false
NIL

$ (AND (EQ 'DOG 'DOG) (< 2 3))                        ;Both comparisons are true
T

$ (AND (ATOM 'DOG) (MEMBER 'DOG '(CAT DOG COW)))                ;Returns last value
(DOG COW)

$ (AND (SYMBOLP 5) (PRINT 'HELLO))                            ;Since test is false,
NIL                                                 ;(PRINT HELLO) is not evaluated
```

OR [$form_1$, $form_2$, ..., $form_n$] Special form

(OR $form_1$ $form_2$ ... $form_n$) evaluates each form in turn until one of them evaluates to a nonNIL value or all the forms are evaluated. If a form is nonNIL, OR returns that nonNIL value, otherwise it returns NIL. The call (OR) returns NIL, the identity for the OR operator. (OR $form_1$ $form_2$ ... $form_n$) is equivalent to the call

(COND ($form_1$) ($form_2$) ... ($form_n$))

Note that OR evaluates a form if and only if all the preceding forms have evaluated to NIL.

```
(DEFMACRO OR LST
  ((NULL LST) NIL)
  (CONS 'COND
      (MAPCAR '(LAMBDA (FORM) (LIST FORM)) LST)) )

$ (OR (EQ 'DOG 'CAT) (< 2 3))                          ;One comparison is true
T

$ (OR (EQ 'DOG 'CAT) (< 3 2))                        ;Neither comparison is true
NIL

$ (SETQ FOO 7)                                                 ;Initialize FOO
7

$ (OR (ZEROP FOO) (/ 35 FOO))                          ;OR returns last value
5

$ (SETQ FOO 0)                                                 ;Initialize FOO
0

$ (OR (ZEROP FOO) (/ 35 FOO))                        ;No "Zero Divide" error break
T
```

5.2 Lists

The functions in this section perform operations on lists including list construction and element selection, substitution, deletion, searching, and sorting.

5.2.1 Selectors

The functions in this subsection are for accessing the elements of lists and the components of trees.

CAR [*object*] Function
CDR [*object*] Function

(CAR *object*) returns the car of *object*, that is, the data object pointed to by the **car cell** of *object*. (CDR *object*) returns the cdr of *object*, that is, the data object pointed to by the **cdr cell** of *object*.

The proper interpretation for the car and the cdr of a data object depends on the data type of the object (see Section 4.1). If the object is a symbol, the car cell points to its current value and the cdr cell points to its property list. If the object is a number, the car cell points to the number itself and the cdr cell points to a symbol indicating its sign and number type. If the object is a list, the car cell points to the first element of the list and the cdr cell points to its tail (i.e. everything but the first element of the list). If the object is a binary tree, the car cell points to its left or **car branch** and the cdr cell points to its right or **cdr branch**.

```
(DEFUN CAR (OBJ)
  (the object pointed to by the car cell of OBJ) )

(DEFUN CDR (OBJ)
  (the object pointed to by the cdr cell of OBJ) )

$ (CAR '(A B C D E))                                    ;Car of a list
A

$ (CDR '(A B C D E))                                    ;Cdr of a list
(B C D E)

$ (CAR '((A . B) . C))                                  ;Car of a binary tree
(A . B)

$ (CDR '((A . B) . C))                                  ;Cdr of a binary tree
C
```

CAAR [*object*] Function
CDAR [*object*] Function
CADR [*object*] Function
CDDR [*object*] Function

CAAAR [*object*] Function
CDAAR [*object*] Function
CADAR [*object*] Function
CDDAR [*object*] Function
CAADR [*object*] Function
CDADR [*object*] Function
CADDR [*object*] Function
CDDDR [*object*] Function

CAAAAR [*object*] Function
CDAAAR [*object*] Function
CADAAR [*object*] Function
CDDAAR [*object*] Function
CAADAR [*object*] Function
CDADAR [*object*] Function
CADDAR [*object*] Function
CDDDAR [*object*] Function
CAAADR [*object*] Function
CDAADR [*object*] Function
CADADR [*object*] Function
CDDADR [*object*] Function
CAADDR [*object*] Function
CDADDR [*object*] Function
CADDDR [*object*] Function
CDDDDR [*object*] Function

Although successive applications of CAR and CDR are sufficient to access any part of a binary tree, all combinations of CAR and CDR up to four compositions are defined in machine language for efficiency and convenience. The name of these functions begins with a C and ends with an R. In between is a sequence of As and Ds indicating the composition of cars and cdrs that the function performs.

```
(DEFUN CDAR (OBJ)
  (CDR (CAR OBJ)) )
```

```
(DEFUN CADDR (OBJ)
  (CAR (CDR (CDR OBJ))) )

$ (SETQ TREE '(                                          ;Initialize TREE
    (((A . B) . (C . D)) . ((E . F) . (G . H))) .
    (((I . J) . (K . L)) . ((M . N) . (O . P)))) )
((((A . B) C . D) (E . F) G . H) ((I . J) K . L) (M . N) O . P)

$ (CAR TREE)                                             ;Car of TREE
(((A . B) C . D) (E . F) G . H)

$ (CDR TREE)                                             ;Cdr of TREE
(((I . J) K . L) (M . N) O . P)

$ (CAAR TREE)                                        ;Car of car of TREE
((A . B) C . D)

$ (CDAR TREE)                                        ;Cdr of car of TREE
((E . F) G . H)

$ (CADR TREE)                                        ;Car of cdr of TREE
((I . J) K . L)

$ (CDDR TREE)                                        ;Cdr of cdr of TREE
((M . N) O . P)

$ (CAAAR TREE)                                   ;Car of car of car of TREE
(A . B)

$ (CADDDR TREE)                              ;Car of cdr of cdr of cdr of TREE
O

$ (CDADAR TREE)                              ;Cdr of car of cdr of car of TREE
F
```

FIRST [*list*] Function
SECOND [*list*] Function
THIRD [*list*] Function
FOURTH [*list*] Function
FIFTH [*list*] Function
SIXTH [*list*] Function
SEVENTH [*list*] Function
EIGHTH [*list*] Function
NINTH [*list*] Function
TENTH [*list*] Function
REST [*list*] Function

These selectors return any one of the first ten elements of a list. REST is included to complement FIRST.

These are the only primitives that refer to the car of a list as the *first* element. All the other primitives (e.g. NTH, SUBLIST, REPLACE-NTH, etc.) refer to the car of a list as the *zeroth* element.

```
(DEFUN SIXTH (LST)
  (NTH 5 LST) )

(DEFUN REST (LST)
  (CDR LST) )

$ (FIRST '(A B C D E))                          ;The first element
A

$ (SECOND '(A B C D E))                      ;The second element of a list
B

$ (THIRD '(A B C D E))                        ;The third element of a list
C

$ (FOURTH '(A B C D E))                      ;The fourth element of a list
D

$ (FIFTH '(A B C D E))                        ;The fifth element of a list
E

$ (REST '(A B C D E))                            ;The rest of a list
(B C D E)
```

LAST [*list*] Function

(LAST *list*) returns the last top-level cons of *list*. Note that LAST returns the last cons, *not* the last element, of *list*. If *list* is an atom, LAST returns NIL. As the final example below shows, the last element can be extracted by taking the CAR of (LAST *list*).

```
(DEFUN LAST (LST)
  ((ATOM LST)  NIL)
  ((ATOM (CDR LST))  LST)
  (LAST (CDR LST)) )

$ (LAST '(A B C D))                                    ;The last cons of a list
(D)

$ (LAST '(A B C . D))                                  ;The last of cons a binary tree
(C . D)

$ (LAST 'FOO)                                          ;The last of a symbol
NIL

$ (LAST 5)                                             ;The last of a number
NIL

$ (CAR (LAST '(A B C)))                                ;The last element of a list
C
```

NTHCDR [*n*, *list*] Function
NTH [*n*, *list*] Function

If *n* is zero or a positive integer, (NTHCDR *n list*) returns the *n*th cdr of *list*. (NTH *n list*) returns the *n*th element of *list*, where the car of *list* is the zeroth element. NTHCDR and NTH return NIL if *n* is neither zero nor a positive integer, or if *list* has *n* or fewer elements.

```
(DEFUN NTHCDR (N LST)
  ((ZEROP N)  LST)
  ((AND (INTEGERP N) (PLUSP N))
    ((ATOM LST)  NIL)
    (NTHCDR (SUB1 N) (CDR LST)) ) )

(DEFUN NTH (N LST)
  ((ATOM (NTHCDR N LST))  NIL)
  (CAR (NTHCDR N LST)) )

$ (NTHCDR 0 '(A B C D))                          ;No cdrs of a list
(A B C D)

$ (NTH 0 '(A B C D))                          ;Zeroth element of a list
A

$ (NTHCDR 2 '(A B C D))                          ;Two cdrs of a list
(C D)

$ (NTH 2 '(A B C D))                          ;Second element of a list
C

$ (NTHCDR 4 '(A B C D))                          ;Four cdrs of a list
NIL

$ (NTH 4 '(A B C D))                       ;No fourth element of a list
NIL

$ (NTHCDR 2 '(A B . C))                          ;Two cdrs of a tree
C
```

LENGTH [*object*] Function
LIST-LENGTH [*object*] Library function

(LENGTH *object*) returns the length of *object* appropriate for its data type. If *object* is NIL or a list, LENGTH returns the number of top-level conses (i.e. elements) in *object*. If *object* is a circular list, LENGTH returns NIL. If *object* is a symbol, LENGTH returns the number of characters in *object*'s print name string. If *object* is a number, LENGTH returns the number of words (a word equals two bytes) required to store *object*'s number vector.

The definition of LIST-LENGTH is *identical* to that of LENGTH and is included in COMMON.LSP solely for compatibility.

```
(DEFUN LENGTH (OBJ)
  ((NULL OBJ)  0)
  ((SYMBOLP OBJ)
    (the number of characters in OBJ's print name string) )
  ((NUMBERP OBJ)
    (CEILING (ABS OBJ) 65536) )
  (ADD1 (LENGTH (CDR OBJ))) )

$ (LENGTH '(A B C D))                                    ;Length of a list
4

$ (LENGTH NIL)                                        ;Length of the empty list
0

$ (LENGTH 'MULISP)                              ;Number of characters in a print name
6

$ (LENGTH 100000)                                 ;Number of words to store a number
2

$ (LENGTH -13)                                    ;Number of words to store a number
1
```

5.2.2 Constructors

The functions in this subsection are for making new data structures (binary trees or linked lists) from simpler structures using conses.

CONS [$object_1$, $object_2$] Function

(CONS $object_1$ $object_2$) returns a cons whose car cell points to $object_1$ and whose cdr cell points to $object_2$. If the value of *FREE-LIST* is a cons, CONS modifies and returns that cons and sets *FREE-LIST* to the cdr of *FREE-LIST* (see Section 5.7.6).

The proper interpretation of (CONS $object_1$ $object_2$) depends on the data type of $object_2$. If $object_2$ is a list, CONS returns a list whose first element is $object_1$ and whose tail is $object_2$. If $object_2$ is an atom or a binary tree, CONS returns a tree whose left or car branch is $object_1$, and whose right or cdr branch is $object_2$. Note that CONS creates a new cons; it does *not* alter either $object_1$ or $object_2$.

```
(DEFUN CONS (OBJ1 OBJ2)
  ((ATOM *FREE-LIST*)
    (a new cons whose car cell points to OBJ1 and whose cdr cell points to OBJ2) )
  (RPLACA *FREE-LIST* OBJ1)
  (RPLACD (PROG1 *FREE-LIST* (POP *FREE-LIST*)) OBJ2) )
```

```
$ (CONS 'A '(B C D))                                    ;Cons a symbol onto a list
(A B C D)

$ (CONS '(A B) '(C D))                                   ;Cons a list onto a list
((A B) C D)

$ (CONS 'A 'B)                                         ;Cons a symbol onto a symbol
(A . B)

$ (CONS '(A B C) 'D)                                     ;Cons a list onto a symbol
((A B C) . D)

$ (CONS 'A NIL)                                           ;Cons a symbol onto NIL
(A)
```

ADJOIN [*object*, *list*, *test*] Function

If *object* is a member of *list* (i.e. there is an *element* of *list* for which (*test object element*) is nonNIL), (ADJOIN *object list test*) returns *list*. Otherwise, ADJOIN returns (CONS *object list*). If the *test* argument is NIL or not given, ADJOIN uses an EQL test.

```
(DEFUN ADJOIN (OBJ LST TEST)
  ((MEMBER OBJ LST TEST)  LST)
  (CONS OBJ LST) )

$ (ADJOIN 'DOG '(CAT COW DOG PIG))                    ;DOG already an element of list
(CAT COW DOG PIG)

$ (ADJOIN 'DOG '(CAT COW RAT PIG))                       ;DOG not an element of list
(DOG CAT COW RAT PIG)

$ (ADJOIN '(A B C) '((R S T) (A B C) (X Y Z)))          ;Not EQL to an element of list
((A B C) (R S T) (A B C) (X Y Z))

$ (ADJOIN '(A B C) '((R S T) (A B C) (X Y Z)) 'EQUAL)     ;EQUAL to an element of list
((R S T) (A B C) (X Y Z))
```

LIST [$object_1$, $object_2$, ..., $object_n$] Function
LIST* [$object_1$, $object_2$, ..., $object_n$] Function

(LIST $object_1$ $object_2$ ... $object_n$) constructs and returns the list consisting of the elements $object_1$ through $object_n$. If called without arguments, LIST returns NIL. LIST can take any number of arguments.

(LIST* $object_1$ $object_2$ ... $object_n$) conses $object_1$ through $object_{n-1}$ onto $object_n$ and returns the resulting list. LIST* is useful for consing several items onto the front of a list. If called with a single argument, LIST* returns that argument.

```
(DEFUN LIST LST
  ((NULL LST)  NIL)
  (CONS (CAR LST) (APPLY 'LIST (CDR LST))) )

(DEFUN LIST* LST
  ((NULL LST)  NIL)
  ((NULL (CDR LST))
    (CAR LST) )
  (CONS (CAR LST) (APPLY 'LIST* (CDR LST))) )

$ (LIST 'A 'B 'C 'D)                                ;Returns a 4 element list
(A B C D)

$ (LIST* 'A 'B 'C 'D)                                 ;Returns a binary tree
(A B C . D)

$ (LIST 'A 'B '(C D))                               ;Returns a 3 element list
(A B (C D))

$ (LIST* 'A 'B '(C D))                              ;Returns a 4 element list
(A B C D)

$ (LIST 'A)                                         ;Returns a 1 element list
(A)

$ (LIST* 'A)                                                ;Returns a leaf
A

$ (LIST)                                             ;Returns the empty list
NIL
```

APPEND [$list_1$, $list_2$, ..., $list_n$] Function

(APPEND $list_1$ $list_2$ ... $list_n$) constructs and returns a list consisting of the elements of the lists $list_1$, $list_2$, through $list_n$. APPEND copies the top-level conses of each of its arguments except the last. If called with a single argument, APPEND simply returns the argument without copying it. Therefore COPY-LIST rather than APPEND must be used to copy a list.

APPEND uses consing to construct and return a new list rather than modifying its arguments. See NCONC in the next subsection for a destructive version of APPEND.

```
(DEFUN APPEND (LST1 LST2)
   ((ATOM LST1)  LST2)
   (CONS (CAR LST1) (APPEND (CDR LST1) LST2)) )

$ (SETQ FOO '(D E F))                                    ;Set FOO to list
(D E F)

$ (APPEND '(A B C) FOO '(G H I))                         ;Append lists
(A B C D E F G H I)

$ FOO                                                    ;FOO unchanged
(D E F)

$ (APPEND '(A B C D) 'E 'F 'G)                           ;Atoms cannot be appended
(A B C D . G)
```

MAKE-LIST [*n*, *object*, *list*] Function
COPY-LIST [*list*] Function
COPY-TREE [*object*] Function

(MAKE-LIST *n object list*) constructs and returns a list of *n* elements, each initialized to *object* concatenated onto *list* as if using NCONC. *n* defaults to 0, *object* and *list* default to NIL.

(COPY-LIST *list*) copies the *top-level* conses of *list*, returning a list that is EQUAL to *list*. Sublists are *not* copied.

(COPY-TREE *object*) copies *all* conses of *object*, returning an object that is EQUAL to *object*.

See COPY-ALIST for copying association lists (see Section 5.2.8).

```
(DEFUN MAKE-LIST (N OBJ LST)
  ((AND (INTEGERP N) (PLUSP N))
    (CONS OBJ (MAKE-LIST (SUB1 N) OBJ LST)) )
  LST )

(DEFUN COPY-LIST (LST)
  ((ATOM LST)  LST)
  (CONS (CAR LST) (COPY-LIST (CDR LST))) )

(DEFUN COPY-TREE (OBJ)
  ((ATOM OBJ)  OBJ)
  (CONS (COPY-TREE (CAR OBJ)) (COPY-TREE (CDR OBJ))) )
```

```
$ (MAKE-LIST 4)                                    ;Make 4 element list
(NIL NIL NIL NIL)

$ (MAKE-LIST 3 '(A B C))                           ;Make 3 element list
((A B C) (A B C) (A B C))

$ (MAKE-LIST 3 5 '(2 3))                           ;Add three 5's to list
(5 5 5 2 3)

$ (COPY-LIST '(A B (C . D) E))                     ;Does not copy (C . D)
(A B (C . D) E)

$ (COPY-TREE '(A B (C . D) E))                     ;Does copy (C . D)
(A B (C . D) E)
```

SUBLIST [*list*, *n*, *m*] Function

If n and m are integers and $0 \leq n \leq m$, (SUBLIST *list n m*) copies and returns the nth through mth elements of *list*, where the car of *list* is the zeroth element. If m is not an integer or is greater than or equal to the length of *list*, m is equivalent to being one less than the length of *list*. If n is not an integer greater than or equal to 0 or if $n > m$, SUBLIST returns NIL.

```
(DEFUN SUBLIST (LST N M)
  ((INTEGERP N)
    ((INTEGERP M)
      (FIRSTN (ADD1 (- M N)) (NTHCDR N LST)) )
    (NTHCDR N LST) ) )

$ (SUBLIST '(A B C D E F) 2 4)                    ;2nd thru 4th element
(C D E)

$ (SUBLIST '(A B C D E F) 2 2)                    ;2nd element
(C)

$ (SUBLIST '(A B C D E F) 2 1)                    ;No elements
NIL

$ (SUBLIST '(A B C D E F) 2)                      ;2nd thru last element
(C D E F)

$ (SUBLIST '(A B C D E F) 0 3)                    ;0th thru 3rd element
(A B C D)
```

FIRSTN [*n*, *list*] Function
BUTLAST [*list*, *n*] Function

If *n* is a positive integer, (FIRSTN *n list*) copies and returns the first *n* elements of *list*. FIRSTN returns NIL if *n* is not a positive integer. If *list* has *n* or fewer elements, FIRSTN copies and returns the elements of *list*.

If *n* is zero or a positive integer, (BUTLAST *list n*) copies and returns all but the last *n* elements of *list*. If *n* is omitted or is neither zero nor a positive integer, BUTLAST copies and returns all but the last element of *list*. See NBUTLAST in the next subsection for a destructive version of BUTLAST.

```
(DEFUN FIRSTN (N LST)
  ((AND (INTEGERP N) (PLUSP N))
    ((ATOM LST) NIL)
    (CONS (CAR LST) (FIRSTN (SUB1 N) (CDR LST))) ) )

(DEFUN BUTLAST (LST N)
  ((AND (INTEGERP N) (>= N 0))
    (FIRSTN (- (LENGTH LST) N) LST) )
  (BUTLAST LST 1) )

$ (FIRSTN 0 '(A B C D))                          ;First 0 elements of a 4 element list
NIL

$ (FIRSTN 2 '(A B C D))                          ;First 2 elements of a 4 element list
(A B)

$ (FIRSTN 5 '(A B C D))                          ;First 5 elements of a 4 element list
(A B C D)

$ (FIRSTN 3 '(A B . C))                            ;First 3 elements of a binary tree
(A B)

$ (BUTLAST '(A B C D E))                                ;But last element of a list
(A B C D)

$ (BUTLAST '(A B C D E) 2)                            ;But last 2 elements of a list
(A B C)

$ (SETQ FOO '(A B C))                                              ;Initialize FOO
(A B C)

$ (BUTLAST FOO)                                             ;But last element of FOO
(A B)

$ FOO                                                        ;FOO remains unchanged
(A B C)
```

LDIFF [*list*, *tail*] Function

If *tail* is a tail (one of the top-level conses) of *list*, (LDIFF *list tail*) copies and returns the elements of *list* up to, but not including, *tail*. If *tail* is not a tail of *list*, LDIFF copies and returns *list*. In neither case is *list* modified or destroyed.

LDIFF is short for list difference.

```
(DEFUN LDIFF (LST TAIL)
  ((ATOM LST)  LST)
  ((EQ LST TAIL)  NIL)
  (CONS (CAR LST) (LDIFF (CDR LST) TAIL)) )

$ (SETQ NUMS '(1 2 3 4 5))                                    ;Initialize NUMS
(1 2 3 4 5)

$ (SETQ TAIL-NUMS (CDDDR NUMS))                               ;Tail of NUMS
(4 5)

$ (LDIFF NUMS TAIL-NUMS)                                      ;But TAIL-NUMS of NUMS
(1 2 3)

$ (LDIFF NUMS '(4 5))                                         ;(4 5) is not a tail of NUMS
(1 2 3 4 5)
```

REVERSE [*list*, *object*] Function
REVAPPEND [*list*, *object*] Library function

(REVERSE *list*) constructs and returns a list consisting of the elements in *list*, but in reverse order. (REVERSE *list object*) returns the reverse of *list* appended to *object*. The result is the same as (APPEND (REVERSE *list*) *object*), but the two argument call on REVERSE is more efficient.

The definition of REVAPPEND is *identical* to that of REVERSE and is included in COMMON.LSP solely for compatibility.

REVERSE uses consing to construct and return a new list rather than modifying *list*. See the next subsection for a destructive version of REVERSE.

```
(DEFUN REVERSE (LST OBJ)
   ((ATOM LST)  OBJ)
   (REVERSE (CDR LST) (CONS (CAR LST) OBJ)) )

$ (REVERSE '(A B C D E))                                    ;Reverse list
(E D C B A)

$ (REVERSE '(A B C) '(D E F))                               ;Reverse and append
(C B A D E F)

$ (REVERSE '(A B C) 'D)                                     ;Reverse and append
(C B A . D)

$ (SETQ FOO '(A B C))                                       ;Initialize FOO
(A B C)

$ (REVERSE FOO)                                             ;Reverse of FOO
(C B A)

$ FOO                                                       ;FOO unchanged
(A B C)
```

OBLIST [ ] Function
STACK-LIST [ ] Function

(OBLIST) constructs and returns the **object list**, a list of the currently active symbols in the system. (It would be more appropriate to call it the **symbol list** but we bow to tradition.) The symbols are listed in the reverse order from which they were read in and/or generated by the string functions: newer symbols to the left of older symbols. Note that unneeded symbols are automatically collected during a **garbage collection** (see Section 4.2).

(STACK-LIST) constructs and returns a list of the objects currently on the **variable stack** (see Section 4.2). The objects are listed in order: more recently pushed objects to the left of less recently pushed objects. STACK-LIST is primarily useful during an error break to help determine what caused the error.

```
(DEFUN OBLIST ()
  (construct and return a list of the symbols currently in the system) )

(DEFUN STACK-LIST ()
  (construct and return a list of the objects on the variable stack) )

$ (LENGTH (OBLIST))                          ;Current number of symbols in system
421

$ (COUNT-IF 'GETD (OBLIST))                  ;Current number of functions in system
365

$ (STACK-LIST)                               ;List of objects on variable stack
(EVAL T DRIVER)
```

5.2.3 Modifiers

The functions in this subsection are for modifying the car or cdr cell of data objects. Modifier functions are used primarily for their *effect* rather than their *value*. By modifying an existing data structure, they can eliminate the need for consing together a whole new structure when only a small part needs to be changed. Since these functions can produce unwanted **side-effects**, they should be used with care.

RPLACA [$object_1$, $object_2$] Function
RPLACD [$object_1$, $object_2$] Function

If $object_1$ is neither the symbol NIL nor a number, (RPLACA $object_1$ $object_2$) replaces the car cell of $object_1$ with a pointer to $object_2$, and (RPLACD $object_1$ $object_2$) replaces the cdr cell of $object_1$ with a pointer to $object_2$. Both functions return the newly modified $object_1$.

The effect of replacing the car or cdr of an object depends on its data type (see Section 4.1). Replacing the car of a list replaces its first element with $object_2$. Replacing the cdr of a list replaces its tail with $object_2$. Replacing the car of a binary tree replaces its left branch with $object_2$. Replacing the cdr of a binary tree replaces its right branch with $object_2$. Replacing the car of a symbol sets its value cell to $object_2$. Replacing the cdr of a symbol sets its property list to $object_2$.

```
(DEFUN RPLACA (OBJ1 OBJ2)
  ((OR (NUMBERP OBJ1) (NULL OBJ1))  OBJ1)
  (replace car cell of OBJ1 with pointer to OBJ2)
  OBJ1 )

(DEFUN RPLACD (OBJ1 OBJ2)
  ((OR (NUMBERP OBJ1) (NULL OBJ1))  OBJ1)
  (replace cdr cell of OBJ1 with pointer to OBJ2)
  OBJ1 )

$ (SETQ FOO '(C A T))                                    ;Initialize FOO
(C A T)

$ (RPLACA FOO 'R)                                        ;Replace car of FOO
(R A T)

$ (RPLACD FOO '(E A D))                                  ;Replace cdr of FOO
(R E A D)

$ FOO                                                    ;Newly modified FOO
(R E A D)
```

FILL [*list*, *object*] Function
REPLACE [$list_1$, $list_2$] Function

By modifying the car of the top-level conses of *list*, (FILL *list object*) replaces the elements of *list* with *object* and returns the newly modified *list*.

By modifying the car of the top-level conses of *list*1, (REPLACE $list_1$ $list_2$) replaces successive elements of $list_1$ with successive elements of $list_2$. If the lists are of unequal lengths, replacement ceases when the end of the shorter list is reached.

```
(DEFUN FILL (LST OBJ)
  ((ATOM LST)  LST)
  (FILL (CDR LST) OBJ)
  (RPLACA LST OBJ) )

(DEFUN REPLACE (LST1 LST2)
  ((OR (ATOM LST1) (ATOM LST2))  LST1)
  (REPLACE (CDR LST1) (CDR LST2))
  (RPLACA LST1 (CAR LST2)) )

$ (SETQ FOO '(X Y Z W))                          ;Initialize FOO
(X Y Z W)

$ (REPLACE FOO '(1 2 3))                    ;Replace 3 elements of FOO
(1 2 3 W)

$ FOO                                             ;Newly modified FOO
(1 2 3 W)

$ (FILL FOO 4)                                     ;Fill FOO with 4's
(4 4 4 4)

$ FOO                                             ;Newly modified FOO
(4 4 4 4)
```

NCONC [$list_1$, $list_2$, ..., $list_n$] Function

(NCONC $list_1$ $list_2$ ... $list_n$) returns a list consisting of the elements of the lists $list_1$, $list_2$, through $list_n$ in this order, by modifying the cdr of the last top-level cons of $list_1$ through $list_{n-1}$.

Note that if NCONC and APPEND, defined in the previous subsection, are given the same arguments, they will return EQUAL lists. However, APPEND uses conses to copy all but its last argument, whereas NCONC actually modifies all but its last argument by redirecting their final cdr cells.

(NCONC *list list*) will turn *list* into a **circular list**. If an attempt is then made to display *list*, the display will continue indefinitely or until halted by a console interrupt (see Section 4.4).

Compare the following examples with those given for APPEND in the previous subsection.

```
(DEFUN NCONC (LST1 LST2)
  ((ATOM LST1)  LST2)
  (RPLACD (LAST LST1) LST2)
  LST1 )

$ (SETQ FOO '(D E F))                                    ;Initialize FOO
(D E F)

$ (NCONC '(A B C) FOO '(G H I))                          ;Concatenate lists
(A B C D E F G H I)

$ FOO                                                    ;Newly modified FOO
(D E F G H I)

$ (NCONC '(A B C D) 'E 'F 'G)                            ;Nconcing atoms
(A B C D . G)
```

NREVERSE [*list, object*] Function
NRECONC [*list, object*] Library function

By modifying the top-level conses of *list*, (NREVERSE *list*) reverses the elements in *list*. (NREVERSE *list object*) returns the nreverse of *list* concatenated to *object*. The result is the same as (NCONC (NREVERSE *list*) *object*), but the two argument call on NREVERSE is more efficient.

The definition of NRECONC is *identical* to that of NREVERSE and is included in COMMON.LSP solely for compatibility.

By destructively modifying *list*, NREVERSE requires no additional conses to reverse *list*. REVERSE is a non-destructive version of NREVERSE (see Section 5.2.2).

```
(DEFUN NREVERSE (LST OBJ)
  ((ATOM LST)  OBJ)
  (NREVERSE (CDR LST) (RPLACD LST OBJ)) )

$ (NREVERSE '(A B C D E))                                    ;Reverse list
(E D C B A)

$ (NREVERSE '(A B C) '(D E F))                              ;Reverse and nconc
(C B A D E F)

$ (NREVERSE '(A B C) 'D)                                    ;Reverse and nconc
(C B A . D)

$ (SETQ FOO '(A B C))                                        ;Initialize FOO
(A B C)

$ (NREVERSE FOO)                                             ;Reverse FOO
(C B A)

$ FOO                                    ;FOO has been destructively modified!
(A)

$ (SETQ FOO '(A B C))                                        ;Initialize FOO
(A B C)

$ (SETQ FOO (NREVERSE FOO))                                  ;Reverse FOO
(C B A)

$ FOO                                                  ;FOO is correctly reversed
(C B A)                                  ;Moral: Always use SETQ with NREVERSE
```

NBUTLAST [*list*, *n*] Function

If *n* is zero or a positive integer, (NBUTLAST *list n*) returns a list consisting of all but the last *n* elements of *list*, by replacing the cdr of the *n*th cons from the end of *list* with NIL. *n* defaults to 1.

By destructively modifying *list*, NBUTLAST requires no additional conses to return all but the last *n* elements of *list*. BUTLAST is a non-destructive version of NBUTLAST (see Section 5.2.2).

```
(DEFUN NBUTLAST (LST N)
  ((AND (INTEGERP N) (>= N 0))
    (NREVERSE (NTHCDR N (NREVERSE LST))) )
  (NBUTLAST LST 1) )

$ (NBUTLAST '(A B C D))                                    ;Remove last element
(A B C)

$ (NBUTLAST '(A B C D) 2)                                  ;Remove last 2 elements
(A B)

$ (SETQ FOO '(A B C))                                      ;Initialize FOO
(A B C)

$ (NBUTLAST FOO)                                           ;Remove last element
(A B)

$ FOO                                                      ;Newly modified FOO
(A B)

$ (NBUTLAST FOO 2)                                         ;Remove last 2 elements
NIL

$ FOO                                                      ;FOO is not modified!
(A B)

$ (SETQ FOO (NBUTLAST FOO 2))                              ;Remove last 2 elements
NIL

$ FOO                                                      ;FOO is now modified!
NIL                                          ;Moral: Always use SETQ with NBUTLAST
```

REPLACE-NTH [*item, list, n*] Function
INSERT-NTH [*item, list, n*] Function
DELETE-NTH [*list, n*] Function

By modifying *list*, (REPLACE-NTH *item list n*) replaces the *n*th element of *list* with *item*, where the car of *list* is the *0*th element of *list*. If *list* has *n* or fewer elements, REPLACE-NTH returns *list*.

By modifying *list*, (INSERT-NTH *item list n*) inserts *item* before the *n*th element of *list*. If *list* has fewer than *n* elements, INSERT-NTH returns *list*.

By modifying *list*, (DELETE-NTH *list n*) deletes the *n*th element of *list*. If *list* has *n* or fewer elements, DELETE-NTH returns *list*.

```
(DEFUN INSERT-NTH (ITEM LST N)
  ((AND (INTEGERP N) (>= N 0))
    ((ZEROP N)
      (CONS ITEM LST) )
    ((ATOM LST)  LST)
    (RPLACD LST (INSERT-NTH ITEM (CDR LST) (SUB1 N))) ) )

(DEFUN DELETE-NTH (LST N)
  ((AND (INTEGERP N) (>= N 0))
    ((ATOM LST)  LST)
    ((ZEROP N)
      (CDR LST) )
    (RPLACD LST (DELETE-NTH (CDR LST) (SUB1 N))) ) )

$ (REPLACE-NTH 'A '(B C D E) 2)                    ;Replace 2nd element
(B C A E)

$ (INSERT-NTH 'A '(B C D E) 2)                     ;Insert before 2nd
(B C A D E)

$ (DELETE-NTH '(A B C D) 1)                        ;Delete 1st element
(A C D)

$ (DELETE-NTH '(A B C D) 4)                        ;Delete 4th element
(A B C D)
```

TCONC [*dotted-pair, object*] Function
LCONC [*dotted-pair, list*] Function

By modifying the list, (TCONC *dotted-pair object*) adds the new element *object* to the end of the list pointed to by the car of *dotted-pair*. TCONC returns *dotted-pair* with the cdr cell modified to point to the new end of the list.

By modifying the list, (LCONC *dotted-pair list*) appends *list* to the end of the list pointed to by the car of *dotted-pair*. LCONC returns *dotted-pair* with the cdr cell modified to point to the new end of the list.

TCONC and LCONC respectively add and append elements to the tail end of a list, efficiently using RPLACD to modify the list. Their first argument is a cons, the car of which points to the beginning of a list and the cdr of which points to the end (i.e. the last cons) of the list. Their second argument is the item to be added to the end of the list. As shown in the examples, the first time they are called, their first argument should be the atom NIL or the list (NIL).

```
(DEFUN TCONC (PAIR OBJ)
  (LCONC PAIR (LIST OBJ)) )

(DEFUN LCONC (PAIR LST)
  ((ATOM LST)  PAIR)
  ((ATOM PAIR)
    (CONS LST (LAST LST)) )
  ((ATOM (CDR PAIR))
    (RPLACA PAIR LST)
    (RPLACD PAIR (LAST LST)) )
  (RPLACD (CDR PAIR) LST)
  (RPLACD PAIR (LAST LST)) )

$ (SETQ FOO (TCONC NIL 'A))                    ;Add A to end of empty list
((A) A)                                        ;Note that SETQ is required

$ (TCONC FOO 'B)                                   ;Add B to end of list
((A B) B)

$ (CAR FOO)                                                ;Extract list
(A B)

$ (SETQ FOO (LIST NIL))                          ;Initialize FOO to (NIL)
(NIL)

$ (LCONC FOO '(A B))                           ;Add (A B) to end of empty list
((A B) B)                                         ;Note: SETQ is not required

$ (LCONC FOO '(C D))                               ;Add (C D) to end of list
((A B C D) D)
```

5.2.4 Deletion

The functions in this subsection are for deleting elements from lists. Both nondestructive and destructive versions are provided.

REMOVE [*item, list, test*] Function
DELETE [*item, list, test*] Function

If *test* is a comparator, (REMOVE *item list test*) returns a top-level copy of *list* except for those elements for which (*test item element*) is nonNIL. *list* is *not* altered.

By modifying the cdr of the top-level conses of *list*, (DELETE *item list test*) deletes all elements of *list* for which (*test item element*) is nonNIL. Although *list* is altered, do *not* assume that *list* is EQ to the value returned by DELETE.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN REMOVE (ITEM LST TEST)
  ((ATOM LST)  LST)
  ((FUNCALL (OR TEST 'EQL) ITEM (CAR LST))
    (REMOVE ITEM (CDR LST) TEST) )
  (CONS (CAR LST) (REMOVE ITEM (CDR LST) TEST)) )

(DEFUN DELETE (ITEM LST TEST)
  ((ATOM LST)  LST)
  ((FUNCALL (OR TEST 'EQL) ITEM (CAR LST))
    (DELETE ITEM (CDR LST) TEST) )
  (delete-aux ITEM LST LST TEST) )

(DEFUN delete-aux (ITEM LST TAIL TEST)
  ((ATOM (CDR TAIL))  LST)
  ((FUNCALL (OR TEST 'EQL) ITEM (CADR TAIL))
    (RPLACD TAIL (CDDR TAIL))
    (delete-aux ITEM LST TAIL TEST) )
  (delete-aux ITEM LST (CDR TAIL) TEST) )
```

```
$ (REMOVE '(2 5) '((5 2) (2 5) (2 3)))                    ;Lists are not EQL
((5 2) (2 5) (2 3))

$ (REMOVE '(2 5) '((5 2) (2 5) (2 3)) 'EQUAL)              ;Lists are EQUAL
((5 2) (2 3))

$ (SETQ ANIMALS '(DOG RAT CAT PIG RAT COW))               ;Initialize ANIMALS
(DOG RAT CAT PIG RAT COW)

$ (DELETE 'RAT ANIMALS)                                    ;Deletes all RATs
(DOG CAT PIG COW)                                        ;and modifies ANIMALS
```

REMOVE-IF [*test*, *list*] Function
DELETE-IF [*test*, *list*] Function

If *test* is a recognizer, (REMOVE-IF *test list*) returns a top-level copy of *list* except for those elements for which (*test element*) is nonNIL. *list* is *not* altered.

By modifying the cdr of the top-level conses of *list*, (DELETE-IF *test list*) deletes all elements of *list* for which (*test element*) is nonNIL. Although *list* is altered, do *not* assume that *list* is EQ to the value returned by DELETE-IF.

```
(DEFUN REMOVE-IF (TEST LST)
  ((ATOM LST)  LST)
  ((FUNCALL TEST (CAR LST))
    (REMOVE-IF TEST (CDR LST)) )
  (CONS (CAR LST) (REMOVE-IF TEST (CDR LST))) )

(DEFUN DELETE-IF (TEST LST)
  ((ATOM LST)  LST)
  ((FUNCALL TEST (CAR LST))
    (DELETE-IF TEST (CDR LST)) )
  (delete-if-aux TEST LST LST) )

(DEFUN delete-if-aux (TEST LST TAIL)
  ((ATOM (CDR TAIL))  LST)
  ((FUNCALL TEST (CADR TAIL))
    (RPLACD TAIL (CDDR TAIL))
    (delete-if-aux TEST LST TAIL) )
  (delete-if-aux TEST LST (CDR TAIL)) )

$ (SETQ NUMS '(-2 0 7 -0.1 3))                          ;Initialize NUMS
(-2 0 7 -0.1 3)

$ (REMOVE-IF 'MINUSP NUMS)                    ;Remove all negative numbers
(0 7 3)

$ NUMS                                                  ;NUMS left unchanged
(-2 0 7 -0.1 3)

$ (DELETE-IF 'MINUSP NUMS)                    ;Delete all negative numbers
(0 7 3)

$ NUMS                                                  ;NUMS now modified
(-2 0 7 3)                                             ;but not EQ to answer
```

REMOVE-DUPLICATES [*list*, *test*] Function
DELETE-DUPLICATES [*list*, *test*] Function

(REMOVE-DUPLICATES *list test*) returns a copy of *list* with all duplicate elements removed. Duplicate elements are those for which (*test element*$_1$ *element*$_2$) is nonNIL.

By modifying the cdr of the top-level conses of *list*, (DELETE-DUPLICATES *list test*) removes duplicate elements of *list*. Duplicate elements are those for which (*test element*$_1$ *element*$_2$) is nonNIL. Although *list* is altered, do *not* assume that *list* is EQ to the value returned by DELETE-DUPLICATES.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN REMOVE-DUPLICATES (LST TEST)
  ((ATOM LST)  LST)
  ((MEMBER (CAR LST) (CDR LST) TEST)
    (REMOVE-DUPLICATES (CDR LST) TEST) )
  (CONS (CAR LST) (REMOVE-DUPLICATES (CDR LST) TEST)) )

(DEFUN DELETE-DUPLICATES (LST TEST)
  ((ATOM LST)  LST)
  ((MEMBER (CAR LST) (CDR LST) TEST)
    (DELETE-DUPLICATES (CDR LST) TEST) )
  (delete-duplicates-aux LST LST TEST) )

(DEFUN delete-duplicates-aux (LST TAIL TEST)
  ((ATOM (CDR TAIL))  LST)
  ((MEMBER (CADR TAIL) (CDDR TAIL) TEST)
    (RPLACD TAIL (CDDR TAIL))
    (delete-duplicates-aux LST TAIL TEST) )
  (delete-duplicates-aux LST (CDR TAIL) TEST) )

$ (SETQ NUMS '(4 2 3 4 5 3 0))                          ;Initialize NUMS
(4 2 3 4 5 3 0)

$ (REMOVE-DUPLICATES NUMS)                              ;Remove duplicates
(2 4 5 3 0)

$ NUMS                                                  ;NUMS left unchanged
(4 2 3 4 5 3 0)

$ (DELETE-DUPLICATES NUMS)                              ;Delete duplicates
(2 4 5 3 0)

$ NUMS                                                  ;NUMS destructively modified
(4 2 4 5 3 0)                                ;Moral: Use SETQ with DELETE-DUPLICATES
```

5.2.5 Search

The functions in this subsection are for searching through the elements of lists to test for certain conditions. If no test is specified, these functions all use an EQL test.

FIND [*object, list, test*] Function
FIND-IF [*test, list*] Function

(FIND *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the test argument is NIL or not given, FIND uses an EQL test.

(FIND-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL.

For both functions, if an element satisfying the test is found, the element is returned; otherwise NIL is returned.

```
(DEFUN FIND (OBJ LST TEST)
  ((ATOM LST)  NIL)
  ((FUNCALL (OR TEST 'EQL) OBJ (CAR LST))
    (CAR LST) )
  (FIND OBJ (CDR LST) TEST) )

$ (FIND 'EAT '(CORN WHEAT OATS RICE) 'FINDSTRING)          ;Element containing EAT
WHEAT

$ (FIND 7 '(5 7 3 8 12 9 3) '<)                  ;First element that 7 is less than
8

$ (SETQ WEIGHT '((SUE 120) (TOM 182) (AL 174)))
((SUE 120) (TOM 182) (AL 174))

$ (FIND 'TOM WEIGHT '(LAMBDA (X Y)(EQ X (CAR Y))))                    ;Tom's weight
(TOM 182)

$ (FIND-IF '(LAMBDA (Y) (EQ 'TOM (CAR Y))) WEIGHT)                    ;Tom's weight
(TOM 182)

$ (FIND-IF 'MINUSP '(4 0 -6 2 -3 7))                ;First negative number in list
-6

$ (FIND-IF '(LAMBDA (X) (MINUSP (CADR X))) '((X 3.7) (Y 0.5) (Z -2.4)))
(Z -2.4)                                          ;First negative coordinate in list
```

POSITION [*object, list, test*] Function
POSITION-IF [*test, list*] Function

(POSITION *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the test argument is NIL or not given, POSITION uses an EQL test.

(POSITION-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL.

For both functions, if an element satisfying the test is found, the zero-based index of the element is returned; otherwise NIL is returned.

```
(DEFUN POSITION (OBJ LST TEST)
  (position-aux OBJ LST TEST 0) )

(DEFUN position-aux (OBJ LST TEST INDEX)
  ((ATOM LST)  NIL)
  ((FUNCALL (OR TEST 'EQL) OBJ (CAR LST))  INDEX)
  (position-aux OBJ (CDR LST) TEST (ADD1 INDEX)) )

$ (POSITION 23 '(2 3 5 7 11 13 17 19 23 29 31 37))          ;Number of primes before 23
8

$ (POSITION '(A B C) '((R S T) (C A B) (A B C)) 'EQUAL)    ;Position of sublist in list
2

$ (POSITION '(A B C) '((R S T) (C A B) (A B C)))    ;Sublist not EQL to element of list
NIL

$ (POSITION-IF 'PLUSP '(-2.5 0 X 3.7 -5.3))         ;Position of first positive element
3

$ (POSITION-IF 'EVENP '(2 3 5 7 11 13))                ;Position of first even element
0
```

COUNT [*object, list, test*] Function
COUNT-IF [*test, list*] Function

(COUNT *object list test*) returns the number of elements in *list* for which (*test object element*) is nonNIL. If the test argument is NIL or not given, COUNT uses an EQL test.

(COUNT-IF *test list*) returns the number of elements in *list* for which (*test element*) is nonNIL.

```
(DEFUN COUNT (OBJ LST TEST)
  (count-aux OBJ LST TEST 0) )

(DEFUN count-aux (OBJ LST TEST COUNTER)
  ((ATOM LST)  COUNTER)
  ((FUNCALL (OR TEST 'EQL) OBJ (CAR LST))
    (count-aux OBJ (CDR LST) TEST (ADD1 COUNTER)) )
  (count-aux OBJ (CDR LST) TEST COUNTER) )

$ (COUNT 'DOG '(CAT DOG COW PIG DOG ANT))                    ;Number of DOGs in list
2

$ (COUNT 7 '(5 8 12 7 9 3) '<)                  ;Number of numbers that 7 is less than
3

$ (COUNT 'T (UNPACK 'SUBSTITUTE))                          ;Number of Ts in SUBSTITUTE
3

$ (COUNT-IF 'EVENP '(3 -6 8 7 0))                       ;Number of even integers in list
3

$ (COUNT-IF '(LAMBDA (CHAR) (MEMBER CHAR '(A E I O U))) (UNPACK 'AUTOMOBILE))
6                                                        ;Number of vowels in AUTOMOBILE
```

MISMATCH [*list*$_1$, *list*$_2$, *test*] Function

(MISMATCH *list*$_1$ *list*$_2$ *test*) compares the elements of *list*$_1$ and *list*$_2$ using the two argument predicate *test* until a mismatch is found (i.e. until *test* returns NIL), or the end of one or both of the lists is reached. If the *test* argument is NIL or not given, MISMATCH uses an EQL test.

If the lists are of equal length and no mismatch is found, MISMATCH returns NIL. If the lists are of unequal length and no mismatch is found, MISMATCH returns the length of the shorter list. Otherwise, MISMATCH returns the zero-based index in the lists of the first pair of elements that fail to match.

```
(DEFUN MISMATCH (LST1 LST2 TEST)
  (mismatch-aux LST1 LST2 TEST 0) )

(DEFUN mismatch-aux (LST1 LST2 TEST INDEX)
  ((AND (ATOM LST1) (ATOM LST2))  NIL)
  ((OR (ATOM LST1) (ATOM LST2))  INDEX)
  ((NOT (FUNCALL (OR TEST 'EQL) (CAR LST1) (CAR LST2)))  INDEX)
  (mismatch-aux (CDR LST1) (CDR LST2) TEST (ADD1 INDEX)) )

$ (MISMATCH '(A B C D E) '(A B D C E))                    ;Index of first mismatch
2

$ (MISMATCH '(A B C D E) '(A B C D))                      ;Length of shorter list
4

$ (MISMATCH '(5 -4 6 9) '(7 -3 9 8) '<)          ;Index of first element greater than
3

$ (MISMATCH '(5 -4 6 2) '(7 -3 9 8) '<)                   ;All elements less than
NIL
```

MEMBER [*object*, *list*, *test*] Function
MEMBER-IF [*test*, *list*] Function

(MEMBER *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the *test* argument is NIL or not given, MEMBER uses an EQL test.

(MEMBER-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL.

For both functions, if an element satisfying the test is found, the tail of *list* beginning with that element is returned; otherwise NIL is returned.

Note that in some versions of LISP, MEMBER uses an EQUAL test. Of course, this effect can easily be achieved by the call (MEMBER *object list* 'EQUAL).

```
(DEFUN MEMBER (OBJ LST TEST)
  ((ATOM LST)  NIL)
  ((FUNCALL (OR TEST 'EQL) OBJ (CAR LST))  LST)
  (MEMBER OBJ (CDR LST) TEST) )

$ (MEMBER 'A '(B C D))                          ;A is not EQL to a member of list
NIL

$ (MEMBER 'A '(B A D))                              ;A is EQL to a member of list
(A D)

$ (MEMBER '(3 4) '((7 8) (3 4) (5 2)))          ;(3 4) is not EQL to a member of list
NIL

$ (MEMBER '(3 4) '((7 8) (3 4) (5 2)) 'EQUAL)    ;(3 4) is EQUAL to a member of list
((3 4) (5 2))

$ (MEMBER-IF 'NUMBERP '(A B 3 C (-7)))                  ;There is a number in list
(3 C (-7))

$ (MEMBER-IF 'MINUSP '(A B 3 C (-7)))           ;There is no negative number in list
NIL
```

5.2.6 Substitution

The functions in this subsection are for substituting new objects for old objects in lists and binary trees. The substitution can be accomplished either by copying or by modifying the original list or binary tree.

SUBSTITUTE [*new, old, list, test*] Function
NSUBSTITUTE [*new, old, list, test*] Function

(SUBSTITUTE *new old list test*) returns a top-level copy of *list*, replacing with *new* the elements of *list* for which (*test old element*) is nonNIL. If the *test* argument is NIL or not given, SUBSTITUTE uses an EQL test. *list* is *not* altered.

By modifying the car of the top-level conses of *list*, (NSUBSTITUTE *new old list test*) replaces with *new* the elements of *list* for which (*test old element*) is nonNIL. *list* is altered.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SUBSTITUTE (NEW OLD LST TEST)
  ((ATOM LST)  LST)
  (CONS (IF (FUNCALL (OR TEST 'EQL) OLD (CAR LST)) NEW (CAR LST))
        (SUBSTITUTE NEW OLD (CDR LST) TEST)) )

(DEFUN NSUBSTITUTE (NEW OLD LST TEST)
  ((ATOM LST)  LST)
  (NSUBSTITUTE NEW OLD (CDR LST) TEST)
  ((FUNCALL (OR TEST 'EQL) OLD (CAR LST))
    (RPLACA LST NEW) )
  LST )

$ (SUBSTITUTE 'CANNIBALS 'NOUN '(NOUN LIKE TO EAT NOUN))                ;Replace NOUNs
(CANNIBALS LIKE TO EAT CANNIBALS)

$ (SETQ EXPN '(+ 3 X (* 2 X) X))                                        ;Initialize EXPN
(+ 3 X (* 2 X) X)

$ (SUBSTITUTE 5 'X EXPN)                                  ;Replace top-level Xs with 5
(+ 3 5 (* 2 X) 5)

$ EXPN                                                          ;EXPN left unchanged
(+ 3 X (* 2 X) X)

$ (NSUBSTITUTE 5 'X EXPN)                                 ;Replace top-level Xs with 5
(+ 3 5 (* 2 X) 5)

$ EXPN                                                           ;EXPN now modified
(+ 3 5 (* 2 X) 5)
```

SUBSTITUTE-IF [*new, test, list*] Function
NSUBSTITUTE-IF [*new, test, list*] Function

(SUBSTITUTE-IF *new test list*) returns a top-level copy of *list*, replacing with *new* all the elements of *list* for which (*test element*) is nonNIL. *list* is *not* altered.

By modifying the car of the top-level conses of *list*, (NSUBSTITUTE-IF *new test list*) replaces with *new* the elements of *list* for which (*test element*) is nonNIL. *list* is altered.

```
(DEFUN SUBSTITUTE-IF (NEW TEST LST)
  ((ATOM LST)  LST)
  (CONS (IF (FUNCALL TEST (CAR LST)) NEW (CAR LST))
        (SUBSTITUTE-IF NEW TEST (CDR LST))) )

(DEFUN NSUBSTITUTE-IF (NEW TEST LST)
  ((ATOM LST)  LST)
  (NSUBSTITUTE-IF NEW TEST (CDR LST))
  ((FUNCALL TEST (CAR LST))
    (RPLACA LST NEW) )
  LST )

$ (SUBSTITUTE-IF 0                    ;Replace top-level small magnitude numbers with 0
                 '(LAMBDA (N) (AND (NUMBERP N) (< -0.1 N 0.1)))
                 '(0.5 -0.09 (1.1 0.03) 0.2))
(0.5 0 (1.1 0.03) 0.2)

$ (SETQ EXPN '(3 -2 (-7 5) -4.7))                                  ;Initialize EXPN
(3 -2 (-7 5) -4.7)

$ (SUBSTITUTE-IF 0 'MINUSP EXPN)                  ;Replace top-level negatives with 0
(3 0 (-7 5) 0)

$ EXPN                                                            ;EXPN left unchanged
(3 -2 (-7 5) -4.7)

$ (NSUBSTITUTE-IF 0 'MINUSP EXPN)                 ;Replace top-level negatives with 0
(3 0 (-7 5) 0)

$ EXPN                                                            ;EXPN now modified
(3 0 (-7 5) 0)
```

SUBST [*new, old, object, test*] Function
NSUBST [*new, old, object, test*] Function

(SUBST *new old object test*) returns a copy of *object*, replacing with *new* all subexpressions of *object* for which (*test old subexpression*) is nonNIL. To prevent unnecessary consing, SUBST uses as much of *object* as possible in the result; however in no case is *object* altered.

By modifying the conses of *object*, (NSUBST *new old object test*) replaces with *new* all subexpressions of *object* for which (*test old subexpression*) is nonNIL. *object* is altered.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SUBST (NEW OLD OBJ TEST)
  ((FUNCALL (OR TEST 'EQL) OLD OBJ)  NEW)
  ((ATOM OBJ)  OBJ)
  (LET ( (LEFT (SUBST NEW OLD (CAR OBJ) TEST))
         (RIGHT (SUBST NEW OLD (CDR OBJ) TEST)) )
    ((AND (EQ LEFT (CAR OBJ)) (EQ RIGHT (CDR OBJ)))  OBJ)
    (CONS LEFT RIGHT) ) )

(DEFUN NSUBST (NEW OLD OBJ TEST)
  ((FUNCALL (OR TEST 'EQL) OLD OBJ)  NEW)
  ((ATOM OBJ)  OBJ)
  (RPLACA OBJ (NSUBST NEW OLD (CAR OBJ) TEST))
  (RPLACD OBJ (NSUBST NEW OLD (CDR OBJ) TEST)) )

$ (SETQ EXPN '(+ 3 X (* 2 X) X))                    ;Initialize EXPN
(+ 3 X (* 2 X) X)

$ (SUBST 5 'X EXPN)                                 ;Replace all Xs with 5
(+ 3 5 (* 2 5) 5)

$ EXPN                                              ;EXPN left unchanged
(+ 3 X (* 2 X) X)

$ (NSUBST 5 'X EXPN)                                ;Replace all Xs with 5
(+ 3 5 (* 2 5) 5)

$ EXPN                                              ;EXPN now modified
(+ 3 5 (* 2 5) 5)
```

SUBST-IF [*new, test, object*] Function
NSUBST-IF [*new, test, object*] Function

(SUBST-IF *new test object*) returns a copy of *object*, replacing with *new* all subexpressions of *object* for which (*test subexpression*) is nonNIL. To prevent unnecessary consing, SUBST-IF uses as much of *object* as possible in the result; however in no case is *object* altered.

By modifying the conses of *object*, (NSUBST-IF *new test object*) replaces with *new* all subexpressions of *object* for which (*test subexpression*) is nonNIL.

```
(DEFUN SUBST-IF (NEW TEST OBJ)
  ((FUNCALL TEST OBJ)  NEW)
  ((ATOM OBJ)  OBJ)
  (LET ( (LEFT (SUBST-IF NEW TEST (CAR OBJ)))
         (RIGHT (SUBST-IF NEW TEST (CDR OBJ))) )
    ((AND (EQ LEFT (CAR OBJ)) (EQ RIGHT (CDR OBJ)))  OBJ)
    (CONS LEFT RIGHT) ) )

(DEFUN NSUBST-IF (NEW TEST OBJ)
  ((FUNCALL TEST OBJ)  NEW)
  ((ATOM OBJ)  OBJ)
  (RPLACA OBJ (NSUBST-IF NEW TEST (CAR OBJ)))
  (RPLACD OBJ (NSUBST-IF NEW TEST (CDR OBJ))) )

$ (SUBST-IF 0                                ;Replace all small magnitude numbers with 0
          '(LAMBDA (N) (AND (NUMBERP N) (< -0.1 N 0.1)))
          '(0.5 -0.09 (1.1 0.03) 0.2))
(0.5 0 (1.1 0) 0.2)

$ (SETQ EXPN '(3 -2 (-7 5) -4.7))                              ;Initialize EXPN
(3 -2 (-7 5) -4.7)

$ (SUBST-IF 0 'MINUSP EXPN)                          ;Replace all negatives with 0
(3 0 (0 5) 0)

$ EXPN                                                         ;EXPN left unchanged
(3 -2 (-7 5) -4.7)

$ (NSUBST-IF 0 'MINUSP EXPN)                         ;Replace all negatives with 0
(3 0 (0 5) 0)

$ EXPN                                                          ;EXPN now modified
(3 0 (0 5) 0)
```

5.2.7 Sets

The functions in this subsection treat lists as if they were sets (i.e. the functions do not consider the order of the elements in lists to be important).

UNION [*list*$_1$, *list*$_2$, *test*] Function
NUNION [*list*$_1$, *list*$_2$, *test*] Library function

(UNION *list*$_1$ *list*$_2$ *test*) constructs and returns a list of those elements that are in either *list*$_1$ *or list*$_2$. If an element is in both lists, it will not be repeated in the result.

NUNION is a destructive version of UNION that modifies the top-level conses of *list*$_1$. NUNION is defined in COMMON.LSP.

For both functions, *test* is a two argument predicate used to determine if elements in *list*$_1$ match elements in *list*$_2$. The first argument of *test* is always from *list*$_1$, the second is always from *list*$_2$. These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN UNION (LST1 LST2 TEST)
  ((ATOM LST1)  LST2)
  ((MEMBER (CAR LST1) LST2 TEST)
    (UNION (CDR LST1) LST2 TEST) )
  (CONS (CAR LST1) (UNION (CDR LST1) LST2 TEST)) )

(DEFUN NUNION (LST1 LST2 TEST)
  ((ATOM LST1)  LST2)
  ((MEMBER (CAR LST1) LST2 TEST)
    (NUNION (CDR LST1) LST2 TEST) )
  (RPLACD LST1 (NUNION (CDR LST1) LST2 TEST)) )

$ (UNION '(LISP FORTH APL) '(APL PASCAL LISP C))
(FORTH APL PASCAL LISP C)
```

INTERSECTION [*list*$_1$, *list*$_2$, *test*] — Function
NINTERSECTION [*list*$_1$, *list*$_2$, *test*] — Library function

(INTERSECTION *list*$_1$ *list*$_2$ *test*) constructs and returns a list of the elements that are in both *list*$_1$ *and list*$_2$. If an element is not in both lists, it will not be included in the result.

NINTERSECTION is a destructive version of INTERSECTION that modifies the top-level conses of *list*$_1$. NINTERSECTION is defined in COMMON.LSP.

For both functions, *test* is a two argument predicate used to determine if elements in *list*$_1$ match elements in *list*$_2$. The first argument of *test* is always from *list*$_1$, the second is always from *list*$_2$. These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN INTERSECTION (LST1 LST2 TEST)
  ((ATOM LST1)  NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (CONS (CAR LST1) (INTERSECTION (CDR LST1) LST2 TEST)) )
  (INTERSECTION (CDR LST1) LST2 TEST) )

(DEFUN NINTERSECTION (LST1 LST2 TEST)
  ((ATOM LST1)  NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (RPLACD LST1 (NINTERSECTION (CDR LST1) LST2 TEST)) )
  (NINTERSECTION (CDR LST1) LST2 TEST) )

$ (INTERSECTION '(3 8 2 0) '(5 2 7 3 4))
(3 2)
```

SET-DIFFERENCE [$list_1$, $list_2$, *test*] Function
NSET-DIFFERENCE [$list_1$, $list_2$, *test*] Library function

(SET-DIFFERENCE $list_1$ $list_2$ *test*) constructs and returns a list of those elements of $list_1$ that do not appear in $list_2$. If an element is in both lists, it will not be included in the result.

NSET-DIFFERENCE is a destructive version of SET-DIFFERENCE that modifies the top-level conses of $list_1$. NSET-DIFFERENCE is defined in COMMON.LSP.

For both functions, *test* is a two argument predicate used to determine if elements in $list_1$ match elements in $list_2$. The first argument of *test* is always from $list_1$, the second is always from $list_2$. These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SET-DIFFERENCE (LST1 LST2 TEST)
  ((ATOM LST1)  NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (SET-DIFFERENCE (CDR LST1) LST2 TEST) )
  (CONS (CAR LST1) (SET-DIFFERENCE (CDR LST1) LST2 TEST)) )

(DEFUN NSET-DIFFERENCE (LST1 LST2 TEST)
  ((ATOM LST1)  NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (NSET-DIFFERENCE (CDR LST1) LST2 TEST) )
  (RPLACD LST1 (SET-DIFFERENCE (CDR LST1) LST2 TEST)) )

$ (SET-DIFFERENCE '(APL PASCAL LISP C) '(LISP FORTH APL))
(PASCAL C)

$ (SET-DIFFERENCE '(4 2 8 5 6) '(3 6 7 2))
(4 8 5)

$ (SET-DIFFERENCE '(3 6 7 2) '(4 2 8 5 6))
(3 7)
```

SET-EXCLUSIVE-OR [*list*$_1$, *list*$_2$, *test*] Function
NSET-EXCLUSIVE-OR [*list*$_1$, *list*$_2$, *test*] Library function

(SET-EXCLUSIVE-OR *list*$_1$ *list*$_2$ *test*) constructs and returns a list of those elements of *list*$_1$ and *list*$_2$ that do not appear in both lists (i.e. those that appear in exactly one list).

NSET-EXCLUSIVE-OR is a destructive version of SET-EXCLUSIVE-OR that modifies the top-level conses of *list*$_1$ and *list*$_2$. NSET-EXCLUSIVE-OR is defined in COMMON.LSP.

For both functions, *test* is a two argument predicate used to determine if elements in *list*$_1$ match elements in *list*$_2$. The first argument of *test* is always from *list*$_1$, the second is always from *list*$_2$. These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SET-EXCLUSIVE-OR (LST1 LST2 TEST)
  (NCONC (SET-DIFFERENCE LST1 LST2 TEST) (SET-DIFFERENCE LST2 LST1 TEST)) )

(DEFUN NSET-EXCLUSIVE-OR (LST1 LST2 TEST
    TMP)
  ((ATOM LST1)  LST2)
  ((MEMBER (CAR LST1) LST2 TEST)
    (SETQ TMP (CAR LST1))
    (NSET-EXCLUSIVE-OR (DELETE TMP LST1 TEST) (DELETE TMP LST2 TEST) TEST) )
  (RPLACD LST1 (SET-DIFFERENCE (CDR LST1) LST2 TEST)) )

$ (SET-EXCLUSIVE-OR '(4 2 8 5 6) '(3 6 7 2))
(4 8 5 3 7)
```

SUBSETP [$list_1$, $list_2$, *test*] Function

If every element of $list_1$ is in $list_2$, (SUBSETP $list_1$ $list_2$ *test*) returns T, otherwise it returns NIL.

test is a two argument predicate used to determine if elements in $list_1$ match elements in $list_2$. The first argument of *test* is always from $list_1$, the second is always from $list_2$. SUBSETP use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SUBSETP (LST1 LST2 TEST)
  ((ATOM LST1))
  ((MEMBER (CAR LST1) LST2 TEST)
    (SUBSETP (CDR LST1) LST2 TEST) ) )

$ (SUBSETP '(A B C) '(D A E C B))                    ;(A B C) is a subset
T
$ (SUBSETP '(A B C) '(D A E F B))                ;(A B C) is not a subset
NIL
```

5.2.8 Association Lists

An **association list** (a-list) is a convenient way to associate indicator keys with data objects. Each element of an a-list is a cons whose car cell points to an **indicator key** and whose cdr cell points to an object. The functions in this subsection are for constructing and using a-lists.

ACONS [*key, object, a-list*] Function
PAIRLIS [*keys, objects, a-list*] Function
COPY-ALIST [*a-list*] Function

(ACONS *key object a-list*) constructs and returns the result of consing the pair (*key . object*) onto the head of *a-list*. ACONS is useful for constructing a-lists.

(PAIRLIS *keys objects a-list*) constructs pairs from the elements of *keys* and *objects* and returns the resulting association list appended onto *a-list*. If the lists are of unequal length, the construction stops when the end of the shorter list is reached. *a-list* defaults to NIL.

(COPY-ALIST *a-list*) copies the top-level conses of *a-list* and the top-level cons of each element of *a-list* that is a cons, returning a list that is EQUAL to *a-list*. To prevent unwanted side-effects, use COPY-ALIST instead of COPY-LIST to copy a-lists.

```
(DEFUN ACONS (KEY OBJ ALST)
  (CONS (CONS KEY OBJ) ALST) )

(DEFUN PAIRLIS (KEYS OBJS ALST)
  ((OR (ATOM KEYS) (ATOM OBJS))  ALST)
  (ACONS (CAR KEYS) (CAR OBJS) (PAIRLIS (CDR KEYS) (CDR OBJS) ALST)) )

(DEFUN COPY-ALIST (ALST)
  ((ATOM ALST)  ALST)
  ((ATOM (CAR ALST))
    (CONS (CAR ALST) (COPY-ALIST (CDR ALST))) )
  (ACONS (CAAR ALST) (CDAR ALST) (COPY-ALIST (CDR ALST))) )

$ (ACONS 'EYES 'BLUE '((WEIGHT . 170) (AGE . 39)))          ;Add pair to a-list
((EYES . BLUE) (WEIGHT . 170) (AGE . 39))

$ (PAIRLIS '(X Y Z) '(5 2 8))                                ;Make an a-list
((X . 5) (Y . 2) (Z . 8))

$ (COPY-ALIST '((A . B) (C . (D E))))                        ;Does not copy (D E)
((A . B) (C . (D E)))
```

ASSOC [*key, a-list, test*] Function
ASSOC-IF [*test, a-list*] Function

(ASSOC *key a-list test*) performs a linear search of *a-list* looking for a pair for which (*test key* (CAR *pair*)) is nonNIL. If the test argument is NIL or not given, ASSOC uses an EQL test. If a pair satisfying the test is found, the pair is returned; otherwise NIL is returned.

(ASSOC-IF *test a-list*) searches *a-list* for a pair for which (*test* (CAR *pair*)) is nonNIL. If a pair satisfying the test is found, the pair is returned; otherwise NIL is returned.

```
(DEFUN ASSOC (KEY ALST TEST)
  ((ATOM ALST)  NIL)
  ((ATOM (CAR ALST))
    (ASSOC KEY (CDR ALST) TEST) )                    ;Skip atoms on a-list
  ((FUNCALL (OR TEST 'EQL) KEY (CAAR ALST))
    (CAR ALST) )                                    ;Return pair if match found
  (ASSOC KEY (CDR ALST) TEST) )                      ;Otherwise, keep searching

(DEFUN ASSOC-IF (TEST ALST)
  ((ATOM ALST)  NIL)
  ((ATOM (CAR ALST))
    (ASSOC-IF TEST (CDR ALST)) )                     ;Skip atoms on a-list
  ((FUNCALL TEST (CAAR ALST))
    (CAR ALST) )                                    ;Return pair if match found
  (ASSOC-IF TEST (CDR ALST)) )                       ;Otherwise, keep searching

$ (SETQ CAPITALS '((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO)))
((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO))          ;Initialize a-list

$ (ASSOC 'FRANCE CAPITALS)                         ;The capital of France is Paris
(FRANCE . PARIS)

$ (ASSOC 'AUSTRALIA CAPITALS)          ;The capital of Australia is not in the a-list
NIL
```

RASSOC [*key, a-list, test*] Function
RASSOC-IF [*test, a-list*] Function

(RASSOC *key a-list test*) performs a linear search of *a-list* looking for a pair for which (*test key* (CDR *pair*)) is nonNIL. If the test argument is NIL or not given, RASSOC uses an EQL test. If a pair satisfying the test is found, the pair is returned; otherwise NIL is returned.

(RASSOC-IF *test a-list*) searches *a-list* for a pair for which (*test* (CDR *pair*)) is nonNIL. If a pair satisfying the test is found, the pair is returned; otherwise NIL is returned. RASSOC is short for reverse associate.

```
(DEFUN RASSOC (KEY ALST TEST)
  ((ATOM ALST)  NIL)
  ((ATOM (CAR ALST))
    (RASSOC KEY (CDR ALST) TEST) )                          ;Skip atoms on a-list
  ((FUNCALL (OR TEST 'EQL) KEY (CDAR ALST))
    (CAR ALST) )                                          ;Return pair if match found
  (RASSOC KEY (CDR ALST) TEST) )                           ;Otherwise, keep searching

(DEFUN RASSOC-IF (TEST ALST)
  ((ATOM ALST)  NIL)
  ((ATOM (CAR ALST))
    (RASSOC-IF TEST (CDR ALST)) )                           ;Skip atoms on a-list
  ((FUNCALL TEST (CDAR ALST))
    (CAR ALST) )                                          ;Return pair if match found
  (RASSOC-IF TEST (CDR ALST)) )                            ;Otherwise, keep searching

$ (SETQ CAPITALS '((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO)))
((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO))                ;Initialize a-list

$ (RASSOC 'PARIS CAPITALS)                              ;Paris is the capital of France
(FRANCE . PARIS)

$ (RASSOC 'CANBERRA CAPITALS)                  ;Canberra is not a capital in the a-list
```

SUBLIS [*a-list, object, test*] Function
NSUBLIS [*a-list, object, test*] Function

(SUBLIS *a-list object test*) returns a copy of *object*, replacing all subexpressions of *object* for which there is a key in *a-list* such that (*test key subexpression*) is nonNIL with the value associated with the key. To prevent unnecessary consing, SUBLIS uses as much of the original object as possible in the result; however in no case is *object* altered.

By modifying the conses of *object*, (NSUBLIS *a-list object test*) replaces all subexpressions of *object* for which there is a key in *a-list* such that (*test key subexpression*) is nonNIL with the value associated with the key. *object* is altered.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SUBLIS (ALST OBJ TEST)
  ((ASSOC OBJ ALST TEST)
    (CDR (ASSOC OBJ ALST TEST)) )
  ((ATOM OBJ)  OBJ)
  (LET ( (LEFT (SUBLIS ALST (CAR OBJ) TEST))
         (RIGHT (SUBLIS ALST (CDR OBJ) TEST)) )
    ((AND (EQ LEFT (CAR OBJ)) (EQ RIGHT (CDR OBJ)))  OBJ)
    (CONS LEFT RIGHT) ) )

(DEFUN NSUBLIS (ALST OBJ TEST)
  ((ASSOC OBJ ALST TEST)
    (CDR (ASSOC OBJ ALST TEST)) )
  ((ATOM OBJ)  OBJ)
  (RPLACA OBJ (NSUBLIS ALST (CAR OBJ) TEST))
  (RPLACD OBJ (NSUBLIS ALST (CDR OBJ) TEST)) )

$ (SETQ EXPN '(* (+ X Y) (^ X 2) Y))                    ;Initialize EXPN
(* (+ X Y) (^ X 2) Y)

$ (SETQ VALS '((X . 5) (Y . 3)))                        ;Initialize VALS
((X . 5) (Y . 3))

$ (SUBLIS VALS EXPN)                                  ;Substitute for X and Y
(* (+ 5 3) (^ 5 2) 3)

$ EXPN                                                   ;EXPN unchanged
(* (+ X Y) (^ X 2) Y)

$ (NSUBLIS VALS EXPN)                                 ;Substitute for X and Y
(* (+ 5 3) (^ 5 2) 3)

$ EXPN                                                  ;Newly modified EXPN
(* (+ 5 3) (^ 5 2) 3)
```

5.2.9 Sorting

The functions in this subsection are for sorting and merging lists.

SPLIT [*list*] Function

(SPLIT *list*) destructively separates *list* into two lists by replacing with NIL the cdr of the top-level cons halfway down *list*. SPLIT returns the tail of *list* beginning where the split was made.

If *list* has an even number of elements, the head and tail of the split will have the same number of elements. If *list* has an odd number of elements, the head will have one more element than the tail.

```
(DEFUN SPLIT (LST)
  ((ATOM LST)  LST)
  (split-aux LST (CDR LST)) )

(DEFUN split-aux (HEAD TAIL)
  ((ATOM TAIL)
    (PROG1 (CDR HEAD) (RPLACD HEAD NIL)) )
  (POP TAIL)
  ((ATOM TAIL)
    (PROG1 (CDR HEAD) (RPLACD HEAD NIL)) )
  (split-aux (CDR HEAD) (CDR TAIL)) )

$ (SETQ NAMES '(TOM SUE JOE PAT SAM))                    ;Set NAMES
(TOM SUE JOE PAT SAM)

$ (SPLIT NAMES)                                  ;SPLIT returns last half
(PAT SAM)

$ NAMES                                          ;NAMES is now first half
(TOM SUE JOE)
```

MERGE [$list_1$, $list_2$, *test*] Function

If *test* is a two argument ordering predicate, (MERGE $list_1$ $list_2$ *test*) destructively merges the elements of $list_1$ and $list_2$ based on *test* and returns the result. (*test* $object_1$ $object_2$) should return NIL if and only if $object_1$ is ordered after $object_2$ or is equal to $object_2$.

MERGE uses a **stable list merge** in the sense that if an element of $list_1$ is equal to an element of $list_2$, the element of $list_1$ will precede that of $list_2$ in the result.

Since MERGE modifies the top-level conses of the lists, it does not require any additional conses. If the lists should not be destroyed, call MERGE with copies of the lists made by the function COPY-LIST.

```
(DEFUN MERGE (LST1 LST2 TEST)
  ((ATOM LST1)  LST2)
  ((ATOM LST2)  LST1)
  ((FUNCALL TEST (CAR LST2) (CAR LST1))
    (RPLACD LST2 (MERGE LST1 (CDR LST2) TEST)) )
  (RPLACD LST1 (MERGE (CDR LST1) LST2 TEST)) )

$ (MERGE '(2 4 6 8 10) '(2 3 5 7 11) '<)                    ;Merge lists of integers
(2 2 3 4 5 6 7 8 10 11)

$ (MERGE '(APPLE ORANGE PEAR) '(BANANA GRAPE PLUM) 'STRING<)   ;Merge lists of symbols
(APPLE BANANA GRAPE ORANGE PEAR PLUM)
```

SORT [*list, test*] Function
STABLE-SORT [*list, test*] Library function

If *test* is a two argument ordering predicate, (SORT *list test*) destructively sorts the elements of *list* based on *test* and returns the result. (*test* $object_1$ $object_2$) should return NIL if and only if $object_1$ is ordered after $object_2$ or is equal to $object_2$.

SORT uses a **list merge sort** requiring time proportional to $n \log(n)$ where n is the length of *list* (see Knuth, The Art of Computer Programming, Volume 3, Addison-Wesley, 1973). The sort is a **stable sort** in the sense that elements of *list* which *test* finds equal to each other will stay in the same order in the result.

Since SORT modifies the top-level conses of *list*, it does not require any additional conses. If *list* should not be destroyed, call SORT with a copy of *list* made by COPY-LIST.

The definition of STABLE-SORT is *identical* to that of SORT and is included in COMMON.LSP solely for compatibility.

```
(DEFUN SORT (LST TEST)
  ((NULL LST) NIL)
  ((NULL (CDR LST)) LST)
  (SETQ LST (CONS (SORT (SPLIT LST) TEST) (SORT LST TEST)))
  (MERGE (CDR LST) (CAR LST) TEST) )

$ (SORT '(5 2 3/4 7 0 3 -4) '<)                    ;Sort list of numbers
(-4 0 0.75 2 3 5 7)

$ (SORT '(SMITH JONES MILLER COX SCHMIDT) 'STRING<)          ;Sort list of names
(COX JONES MILLER SCHMIDT SMITH)
```

5.3 Symbols

A **symbol** is a primitive data object having a value, a property list, a print name, and a function definition (see Section 4.1.1).

5.3.1 Value

The primitives in this subsection are for accessing and modifying the value of symbols by modifying their value cell. They can be used to assign values to program variables. If a variable is not a **local variable** (i.e. not a formal parameter of the currently executing function), assignments to the variable will remain in effect even after the function is exited. As with the modifier functions (see Section 5.2.3), this phenomenon is called a **side-effect** of the function.

SYMBOL-VALUE [*symbol*] Library function
BOUNDP [*symbol*] Library function

If *symbol* is a symbol, (SYMBOL-VALUE *symbol*) returns *symbol*'s value (i.e. the data object pointed to by *symbol*'s value cell). Otherwise, SYMBOL-VALUE returns NIL.

If *symbol* is NIL or a symbol having a value other than itself, (BOUNDP *symbol*) returns T, otherwise it returns NIL.

Since the **muLISP** interpreter automatically evaluates symbols, explicit use of SYMBOL-VALUE and BOUNDP are rarely needed. They are defined in COMMON.LSP.

```
(DEFUN SYMBOL-VALUE (SYM)
  ((SYMBOLP SYM)
    (CAR SYM) ) )

(DEFUN BOUNDP (SYM)
  ((SYMBOLP SYM)
    ((NULL SYM))
    (NEQ SYM (CAR SYM)) ) )

$ (BOUNDP 'NUM)                                    ;Test if NUM is bound
NIL

$ (SETQ NUM 5)                                      ;Assign NUM a value
5

$ (BOUNDP 'NUM)                                    ;Test if NUM is bound
T

$ (SYMBOL-VALUE 'NUM)                               ;The value of NUM
5
```

SET [*symbol, object*] Function

If *symbol* is a nonNIL symbol, (SET *symbol object*) replaces the value cell of *symbol* with a pointer to *object* and returns *object*. If *symbol* is NIL, SET returns object without changing the value of NIL. (Note that the value of the symbol T is *not* protected.) If *symbol* is not a symbol, a Nonsymbolic Argument error break occurs.

RPLACA and SET are identical as far as their effect on their first argument. However, RPLACA returns its first argument, whereas SET returns its second argument. As a general rule, SET should be used if the first argument is a symbol; RPLACA should be used if the first argument is a cons.

```
(DEFUN SET (SYM OBJ)
  ((SYMBOLP SYM)
    ((NULL SYM)  OBJ)
    (RPLACA SYM OBJ)
    OBJ )
  (BREAK (LIST 'SET SYM OBJ) "Nonsymbolic Argument") )

$ (SET 'FOO '(A B C))                                    ;Assign FOO a value
(A B C)

$ FOO                                                    ;New value of FOO
(A B C)

$ (SET 'PET 'DOG)                                        ;Assign PET a value
DOG

$ (SET PET 'ROVER)                          ;Assign the value of PET (i.e. DOG)
ROVER                                                              ;a value

$ DOG                                                    ;The value of DOG
ROVER

$ PET                                                    ;The value of PET
DOG
```

SETQ [$symbol_1$, $form_1$, ..., $symbol_n$, $form_n$] Special form

(SETQ *symbol form*) evaluates *form*, sets the value of *symbol* to result, and returns the result. If SETQ is given more than two arguments, the form evaluation and assignments are made sequentially. If there are an odd number of arguments, the last symbol is set to NIL. SETQ returns the value assigned to the last symbol.

Note that SETQ is a special form and that its first argument is *not* evaluated. If the first argument is not a symbol, a Nonsymbolic Argument error break occurs.

SETQ is short for SET quote. Although SET is more primitive, SETQ is used far more often than SET because assignments are usually made to a variable rather than to the value of a variable.

```
(DEFMACRO SETQ (SYM OBJ)
  (LIST 'SET (LIST 'QUOTE SYM) OBJ) )

$ (SETQ FOO '(A B C))                                    ;Assign FOO to list
(A B C)

$ FOO                                                    ;Current value of FOO
(A B C)

$ (SETQ FOO (CDR FOO))                                   ;Assign FOO to cdr of list
(B C)

$ FOO                                                    ;Current value of FOO
(B C)

$ (SETQ NUM 5)                                           ;Assign NUM to 5
5

$ (SETQ NUM (+ 3 4) SQR (* NUM NUM))                     ;Assigns NUM to 7 and SQR to 7*7
49

$ NUM                                                    ;The value of NUM is now 7
7

$ SQR                                                    ;The value of SQR is 49 (not 25)
49
```

PSETQ [$symbol_1$, $form_1$, ..., $symbol_n$, $form_n$] Special form

(PSETQ $symbol_1$ $form_1$ ... $symbol_n$ $form_n$) evaluates $form_1$ through $form_n$, simultaneously sets the value of $symbol_1$ through $symbol_n$ to the results, and returns NIL. Note that PSETQ is a special form and that the symbols are *not* evaluated. If any $symbol_i$ is not a symbol, PSETQ generates a Nonsymbolic Argument error break.

The effect of PSETQ on its arguments is identical to that of SETQ except that all the forms are evaluated before making any assignments to symbols. PSETQ is short for parallel SETQ.

As the last example shows, PSETQ can be used to efficiently interchange the value of two variables without having to introduce an intermediary.

```
$ (SETQ NUM 5)                                          ;Assign NUM to 5
5

$ (PSETQ NUM (+ 3 4) SQR (* NUM NUM))                ;Assigns NUM to 7 and SQR to 5*5
NIL

$ NUM                                                     ;The value of NUM is now 7
7

$ SQR                                                ;The value of SQR is 25 (not 49)
25

$ (PSETQ NUM SQR SQR NUM)                                ;Swap value of NUM & SQR
NIL

$ NUM                                                     ;Value of NUM is now 25
25

$ SQR                                                      ;Value of SQR is now 7
7
```

SETF [$place_1$, $form_1$, ..., $place_n$, $form_n$] Library macro

If *place* is a valid place form, (SETF *place form*) evaluates *form*, stores the result in *place*, and returns the result. If SETF is given more than two arguments, the form evaluation and assignments are made sequentially. If there are an odd number of arguments, the last place form is set to NIL. SETF returns the value assigned to the last place form.

If the first argument of SETF is not a valid place form an Invalid Place Form error break occurs. The following are the valid **place forms**:

- symbols,
- lists beginning with one of these selector functions: CAR, CDR, CAAR, CADR, ..., CDDDDR, FIRST, SECOND, THIRD, ..., TENTH, REST, NTH, GET, MACRO-FUNCTION, SYMBOL-VALUE, SYMBOL-PLIST, and SYMBOL-FUNCTION, and
- macro calls that expand into valid place forms.

SETF is defined in COMMON.LSP.

```
(DEFMACRO SETF LST
  ((NULL LST)  NIL)
  ((NULL (CDDR LST))
    ((ATOM (CAR LST))
      (CONS 'SETQ LST) )
    (MAKE-SET-FORM (CAR LST) (CADR LST)) )
  (CONS 'PROGN (setf-aux LST)) )

(DEFUN setf-aux (LST)
  ((NULL LST)  NIL)
  ((ATOM (CAR LST))
    (CONS (LIST 'SETQ (CAR LST) (CADR LST)) (setf-aux (CDDR LST))) )
  (CONS (MAKE-SET-FORM (CAR LST) (CADR LST)) (setf-aux (CDDR LST))) )

$ (SETQ FOO '(A B C))                                    ;Initialize FOO
(A B C)

$ (SETF (SECOND FOO) 5)                          ;2nd element of FOO set to 5
5

$ FOO                                                    ;Newly modified FOO
(A 5 C)
```

POP [*symbol*] Special form

(POP *symbol*) returns the "top" (i.e car) of the **stack** (i.e. list) named *symbol* and updates the value of *symbol* to the rest (i.e. cdr) of the stack. If *symbol* is not a symbol, a Nonsymbolic Argument error break occurs. If the value of *symbol* is not a cons, POP returns NIL.

This special form provides the LISP analog of the machine language pop instruction used to fetch information from the top of a stack.

```
(DEFMACRO POP (SYM)
  (LIST 'PROG1
        (LIST 'CAR SYM)
        (LIST 'SETQ SYM (LIST 'CDR SYM))) )

$ (SETQ STACK-LIST '(A B C D E F))                    ;Initialize stack
(A B C D E F)

$ (POP STACK-LIST)                                    ;Pop A off stack
A

$ (POP STACK-LIST)                                    ;Pop B off stack
B

$ (POP STACK-LIST)                                    ;Pop C off stack
C

$ STACK-LIST                                          ;Shortened stack
(D E F)
```

PUSH [*form, symbol*] Special form
PUSHNEW [*form, symbol, test*] Library macro

(PUSH *form symbol*) evaluates *form* and pushes (i.e. conses) the result onto the "stack" named *symbol* and updates the value of *symbol* to the larger stack. PUSH returns the enlarged stack (i.e. list). If *symbol* is not a symbol, a Nonsymbolic Argument error break occurs. This special form provides the LISP analog of the machine language push instruction used to save information on top of a stack.

If the value of *form* is *not* a member of the "stack" named *symbol*, (PUSHNEW *form symbol test*) pushes (i.e. conses) the value of *form* onto the stack, updates the value of *symbol* to the larger stack, and returns the enlarged stack. Otherwise, PUSHNEW simply returns the stack. If *symbol* is not a symbol, a Nonsymbolic Argument error break occurs. PUSHNEW is defined in COMMON.LSP.

```
(DEFMACRO PUSH (OBJ SYM)
  (LIST 'SETQ SYM (LIST 'CONS OBJ SYM)) )

(DEFMACRO PUSHNEW (OBJ SYM TEST)
  (LIST 'SETQ SYM (LIST 'ADJOIN OBJ SYM TEST)) )

$ (SETQ STACK-LIST NIL)                                    ;Empty stack
NIL

$ (PUSH 'A STACK-LIST)                                     ;Push A on stack
(A)

$ (PUSH 'B STACK-LIST)                                     ;Push B on stack
(B A)

$ (PUSHNEW 'C STACK-LIST)                                  ;Push C on stack
(C B A)

$ (PUSHNEW 'B STACK-LIST)                                  ;B already on stack
(C B A)

$ STACK-LIST                                               ;Current stack
(C B A)
```

5.3.2 Property List

The functions in this subsection are for managing the properties and flags on the property lists of symbols. They aid the construction of dynamically extensible data bases from which information can be rapidly extracted or modified.

A symbol's **property list** is an a-list (see Section 5.2.8) used to store the properties and flags associated with the symbol. Since flags are atoms and properties are conses, they are easily distinguishable from one another on property lists.

PUT [*symbol, key, object*] Function

(PUT *symbol key object*) places *object* on the property list of *symbol* under the indicator *key*.

Since PUT uses an EQUAL test when searching the property list for a match to *key*, property keys need not be atomic. Note that any previous value under the indicator *key* is automatically discarded.

Since PUT uses a RPLACD operation to add a property value to a property list, care should used when sharing property lists.

```
(DEFUN PUT (SYM KEY OBJ)
  ((OR (NULL SYM) (NOT (SYMBOLP SYM)))  NIL)
  ((ASSOC KEY (CDR SYM))
    (RPLACD (ASSOC KEY (CDR SYM)) OBJ)
    OBJ )
  (RPLACD SYM (ACONS KEY OBJ (CDR SYM)))
  OBJ )

$ (PUT 'USA 'POPULATION 250000000)                    ;Store population of USA
250000000

$ (PUT 'FRANCE 'POPULATION 57000000)                 ;Store population of FRANCE
57000000

$ (PUT 'JAPAN 'POPULATION 125000000)                  ;Store population of JAPAN
125000000

$ (PUT 'USA 'AREA 3536855)                  ;Store area of USA (in square miles)
3536855

$ (PUT 'FRANCE 'AREA 211208)                              ;Store area of FRANCE
211208

$ (PUT 'JAPAN 'AREA 142726)                               ;Store area of JAPAN
142726
```

GET [*symbol, key, default*] Function
GETF [*plist, key, default*] Library function
GET-PROPERTIES [*plist, key-list*] Library function
SYMBOL-PLIST [*symbol*] Library function

(GET *symbol key default*) returns the property value associated with *symbol* under the indicator *key*. If no such value exists, GET returns *default* or NIL if no default is given.

(GETF *plist key default*) returns the property value on the property list *plist* under the indicator *key*. If no such value exists, GETF returns *default* or NIL if no default is given. It is defined in COMMON.LSP.

(GET-PROPERTIES *plist key-list*) returns the first property (*key* . *value*) associated with any indicator in *key-list* on the property list *plist*. It is defined in COMMON.LSP.

If *symbol* is a symbol, (SYMBOL-PLIST *symbol*) returns *symbol*'s property list (i.e. the data object pointed to by *symbol*'s property list cell). Otherwise, SYMBOL-PLIST returns NIL. It is defined in COMMON.LSP.

To run the following examples, first run the examples for PUT on the previous page.

```
(DEFUN GET (SYM KEY)
  (CDR (ASSOC KEY (CDR SYM))) )

(DEFUN GET-PROPERTIES (LST1 LST2)
  (ASSOC-IF '(LAMBDA (PAIR) (MEMBER (CAR PAIR) LST2)) LST1) )

(DEFUN SYMBOL-PLIST (SYM)
  ((SYMBOLP SYM)
    (CDR SYM) ) )

$ (SYMBOL-PLIST 'USA)                                    ;Property list of USA
((AREA . 3536855) (POPULATION . 250000000))

$ (GET 'USA 'POPULATION)                                 ;Population of USA
250000000

$ (GET 'AUSTRALIA 'POPULATION)                ;Population of Australia is not stored
NIL

$ (DEFUN POPULATION-DENSITY (COUNTRY)          ;Define a population density function
    (/ (GET COUNTRY 'POPULATION)
       (GET COUNTRY 'AREA)) )
POPULATION-DENSITY

$ (POPULATION-DENSITY 'FRANCE)                      ;French persons per square mile
269.8761408

$ (POPULATION-DENSITY 'JAPAN)                      ;Japanese persons per square mile
875.8039867
```

REMPROP [*symbol, key*] Function
REMF [*place, key*] Library macro

(REMPROP *symbol key*) removes from *symbol*'s property list the property associated with *key*. REMPROP returns the old property value if one was found, otherwise it returns NIL.

REMF is the generalized variable form of REMPROP (see SETF in Section 5.3.1).

Since REMPROP and REMF use a destructive RPLACD operation to remove an element from a property list, care should used when sharing property lists.

To run the following examples, first run the examples for PUT given on a previous page.

```
(DEFUN REMPROP (SYM KEY)
  ((ATOM (CDR SYM)) NIL)
  ((EQUAL (CAADR SYM) KEY)
    (SETQ KEY (CDADR SYM))
    (RPLACD SYM (CDDR SYM))
    KEY )
  (REMPROP (CDR SYM) KEY) )

$ (REMPROP 'USA 'AREA)                          ;Remove property
3536855

$ (GET 'FRANCE 'AREA)                           ;France's area
211208

$ (GET 'USA 'AREA)                              ;USA's area is unknown
NIL
```

FLAG [*symbol, flag*] Function
FLAGP [*symbol, flag*] Function
REMFLAG [*symbol, flag*] Function

If *symbol* is a nonNIL symbol, (FLAG *symbol flag*) adjoins *flag* to *symbol*'s property list, and returns *flag*. Otherwise, it returns NIL.

If *flag* is a member of *symbol*'s property list, (FLAGP *symbol flag*) returns a nonNIL value (specifically the property list beginning with *flag*). Otherwise, it returns NIL.

If *flag* is a member of *symbol*'s property list, (REMFLAG *symbol flag*) removes *flag* from the property list, and returns T. Otherwise, it returns NIL.

Flags must be atomic data objects (normally symbols) so the property list functions can distinguish them from properties on property lists.

```
(DEFUN FLAG (SYM FLAG)
  ((OR (NULL SYM) (NOT (SYMBOLP SYM)))  NIL)
  (RPLACD SYM (ADJOIN FLAG (CDR SYM) 'EQUAL))
  FLAG )

(DEFUN FLAGP (SYM FLAG)
  (MEMBER FLAG (CDR SYM) 'EQUAL) )

(DEFUN REMFLAG (SYM FLAG)
  ((ATOM (CDR SYM))  NIL)
  ((EQUAL FLAG (CADR SYM))
    (RPLACD SYM (CDDR SYM))
    T )
  (REMFLAG (CDR SYM) FLAG) )

$ (FLAG 'JAN 'MALE)                                    ;Flag Jan to be male
MALE

$ (FLAGP 'JAN 'MALE)                                  ;Jan is flagged as male
(MALE)

$ (FLAGP 'SAM 'MALE)                                ;Sam is not flagged as male
NIL

$ (FLAG 'JAN 'TALL)                                    ;Flag Jan to be tall
TALL

$ (REMFLAG 'JAN 'MALE)                                ;Remove Jan's male flag
T

$ (FLAGP 'JAN 'MALE)                              ;Jan is no longer flagged as male
NIL
```

5.3.3 Definition

The primitives in this subsection are for managing the function definition cell associated with each symbol. When a function is defined, the definition is automatically transformed into an extremely dense form called **pseudo-code**. The inverse process of transforming a definition into a linked list is also automatic.

GETD [*symbol, flag*]	Function
SPECIAL	Keyword
LAMBDA	Keyword
NLAMBDA	Keyword
MACRO	Keyword

If *symbol* is a user-defined function, (GETD *symbol*) decompiles and returns the definition of *symbol* in linked list form. If *symbol* is a pre-defined function or special form, (GETD *symbol*) returns the physical memory address (in the 8086 code segment) of the **machine language** routine. GETD returns the offset of the routine in the code segment. If *symbol* is not a function or special form, (GETD *symbol*) returns NIL.

If *flag* is nonNIL, (GETD *symbol flag*) returns SPECIAL if *symbol* is a special form, LAMBDA if *symbol* is an eval function, NLAMBDA if *symbol* is a no-eval function, MACRO if *symbol* is a macro, or NIL if *symbol* is undefined. See Section 5.7.1 for an explanation of the different types of functions.

If a program only needs to know if *symbol* is a defined function, it is faster to call GETD with *flag* nonNIL since no consing is required.

```
(DEFUN GETD (SYM FLAG)
  ((NOT (SYMBOLP SYM))  NIL)
  ((undefined SYM)  NIL)
  ((NULL FLAG)
    ((machine language routine SYM)
      (the memory address of the routine) )
    (the S-expression equivalent of the pseudo-code definition of SYM) )
  ((if SYM is a special form)  'SPECIAL)
  ((if SYM is an eval function)  'LAMBDA)
  ((if SYM is a no-eval function)  'NLAMBDA)
  ((if SYM is a macro)  'MACRO) )

$ (GETD 'CONS)                                    ;Address of CONS routine
2673                                    ;result depends on version of muLISP

$ (GETD 'CONS T)                                  ;CONS is an eval function
LAMBDA
```

SYMBOL-FUNCTION [*symbol*] Library function

If *symbol* is a defined function, (SYMBOL-FUNCTION *symbol*) returns *symbol*'s definition (i.e. the decompiled pseudo-code or the address of the machine code pointed to by *symbol*'s function definition cell). If *symbol* is undefined, an Undefined Function error break occurs.

SYMBOL-FUNCTION is defined in COMMON.LSP.

```
(DEFUN SYMBOL-FUNCTION (SYM)
  ((SYMBOLP SYM)
    ((GETD SYM T)
      (GETD SYM) )
    (BREAK (LIST 'SYMBOL-FUNCTION SYM) "Undefined Function") ) )

$ (SYMBOL-FUNCTION 'FBOUNDP)                    ;FBOUNDP's definition
(LAMBDA (SYM) (GETD SYM T))
```

EVAL-FUNCTION-P [*symbol*] Library function
NO-EVAL-FUNCTION-P [*symbol*] Library function
MACRO-P [*symbol*] Library function
SPECIAL-FORM-P [*symbol*] Library function
FBOUNDP [*symbol*] Library function

These predicates return T if *symbol* is a function of the given type, otherwise they return NIL. EVAL-FUNCTION-P tests for eval functions. NO-EVAL-FUNCTION-P tests for no-eval functions. MACRO-P tests for macros. SPECIAL-FORM-P tests for special forms. FBOUNDP tests for functions of any of these types.

These functions are defined in COMMON.LSP.

```
(DEFUN EVAL-FUNCTION-P (SYM)
  (EQ (GETD SYM T) 'LAMBDA) )

(DEFUN NO-EVAL-FUNCTION-P (SYM)
  (EQ (GETD SYM T) 'NLAMBDA) )

(DEFUN MACRO-P (SYM)
  (EQ (GETD SYM T) 'MACRO) )

(DEFUN SPECIAL-FORM-P (SYM)
  (EQ (GETD SYM T) 'SPECIAL) )

(DEFUN FBOUNDP (SYM)
  (GETD SYM T) )
```

```
$ (EVAL-FUNCTION-P 'APPEND)                    ;APPEND is a function
T

$ (MACRO-P 'DO)                                ;DO is a macro
T

$ (SPECIAL-FORM-P 'IF)                         ;IF is a special form
T

$ (FBOUNDP NIL)                                ;NIL is not a defined function
NIL
```

PUTD [*symbol, definition*] Function

If *definition* is a lambda-list, (PUTD *symbol definition*) compiles *definition* into pseudo-code, modifies *symbol*'s function definition cell to point to the pseudo-code, and returns *symbol*. Quoted nonatomic objects (i.e. **constants**) within a function definition are not compiled but remain stored as a linked list. Therefore, the interpreter does not have to decompile such constants each time they are encountered, and constants can contain conses with nonNIL atomic cdr cells.

If *definition* is zero or a positive integer less than 65536, (PUTD *symbol definition*) modifies *symbol*'s function cell to *definition*, and returns *symbol*. This makes it possible to link to user-defined machine language routines (see Section 9.4).

The variable *CONDENSE* controls the condensing of pseudo-code while compiling. The variable MACROEXPAND controls the expansion of macros while compiling pseudo-code (see Section 5.7.5).

```
(DEFUN PUTD (SYM DEFN)
  ((OR (NOT (SYMBOLP SYM)) (SYMBOLP DEFN)) NIL)
  ((NUMBERP DEFN)
    ((AND (INTEGERP DEFN) (PLUSP DEFN))
      (store DEFN in SYM's function cell)
      SYM ) )
  (compile DEFN into pseudo-code)
  (store a pointer to the pseudo-code in SYM's
      function cell)
  SYM )

$ (PUTD 'SQUARE                                    ;Define a function
        '(LAMBDA (NUM) (* NUM NUM)) )
SQUARE

$ (SQUARE 5)                                       ;Call the function
25
```

MOVD [$symbol_1$, $symbol_2$] Function

(MOVD $symbol_1$ $symbol_2$) copies $symbol_1$'s function definition cell to $symbol_2$'s and returns $symbol_2$. In cases where MOVD is sufficient, it should be used instead of a combination of GETD and PUTD since no pseudo-code compilation is required.

The examples show how MOVD can be used to make more mnemonic symbols for some of the primitive LISP selector functions without sacrificing execution efficiency.

```
(DEFUN MOVD (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (replace the contents of SYM2's function
       cell with the contents of SYM1's function cell)
    SYM2 ) )

$ (MOVD 'CAR 'FIRST)                                    ;Define FIRST
FIRST

$ (MOVD 'CDR 'REST)                                     ;Define REST
REST

$ (MOVD 'CADR 'SECOND)                                  ;Define SECOND
SECOND

$ (MOVD 'CADDR 'THIRD)                                  ;Define THIRD
THIRD

$ (MOVD 'CADDDR 'FOURTH)                                ;Define FOURTH
FOURTH

$ (FIRST '(A B C D))                                    ;Call FIRST
A
```

REMD [*symbol*] Function
FMAKUNBOUND [*symbol*] Library function

(REMD *symbol*) sets *symbol*'s function definition cell to point to a machine language routine that generates an Undefined Function error break when *symbol* is called as a function (see Section 5.10.3). REMD returns *symbol*.

The definition of FMAKUNBOUND is *identical* to that of REMD and is included in COMMON.LSP solely for compatibility.

The primary use of REMD is to make symbols be undefined so they can be collected by the garbage collector (see Section 4.2.2).

```
(DEFUN REMD (SYM)
  ((SYMBOLP SYM)
    (MOVD 'UNDEFINED SYM) ) )

$ (DEFUN RCONS (SYM1 SYM2) (CONS SYM2 SYM1))                ;Define RCONS function
RCONS

$ (RCONS 'DOG 'CAT)                                                ;Call RCONS
(CAT . DOG)

$ (REMD 'RCONS)                                    ;Remove RCONS's definition
RCONS

$ (GETD 'RCONS)                                          ;RCONS now undefined
NIL

$ (RCONS 'DOG 'CAT)                                                ;Call RCONS
Undefined Function Break: (RCONS 'DOG 'CAT)
```

DEFUN [*symbol, arglist, form*$_1$, ..., *form*$_n$] Special form
DEFUN [*symbol, (ftype arglist form*$_1$ ... *form*$_n$)] Special form
DEFUN [*symbol, address*] Special form

The function defining special form DEFUN can be called in either of three ways: The first way is for defining eval functions. The second way is for defining no-eval functions and macros. The third way is for linking to user-defined machine language routines.

(DEFUN *symbol arglist form*$_1$ ... *form*$_n$) compiles *arglist* and the forms into pseudo-code assuming an eval or LAMBDA function type. DEFUN then modifies *symbol*'s function definition cell to point to the resulting pseudo-code, and returns *symbol*.

If *ftype* is the symbol LAMBDA, NLAMBDA, or MACRO, the call (DEFUN *symbol* (*ftype arglist form*$_1$ ... *form*$_n$)) compiles *arglist* and the forms into pseudo-code and makes *symbol* an eval function, a no-eval function, or a macro depending on *ftype*. DEFUN then modifies *symbol*'s function cell to point to the resulting pseudo-code, and returns *symbol*.

If *address* is zero or a positive integer less than 65536, (DEFUN *symbol address*) modifies *symbol*'s function cell to *address*, and returns *symbol*. See Section 9.4 for details on linking to machine language routines.

Although it is defined in terms of the more primitive function PUTD, DEFUN is usually a more convenient means of defining functions. Examples of DEFUN appear throughout this chapter.

```
(DEFMACRO DEFUN (SYM . BODY)
  ((ATOM BODY)  NIL)
  ((NUMBERP (CAR BODY))
    (LIST 'PUTD (LIST 'QUOTE SYM) (CAR BODY)) )
  ((MEMBER (CAAR BODY) '(LAMBDA NLAMBDA MACRO))
    (LIST 'PUTD (LIST 'QUOTE SYM) (CONS 'QUOTE BODY)) )
  (LIST 'PUTD (LIST 'QUOTE SYM) (LIST 'QUOTE (CONS 'LAMBDA BODY))) )
```

DEFMACRO [*symbol, arglist, form*$_1$, ..., *form*$_n$] Special form

(DEFMACRO *symbol arglist form*$_1$ *form*$_2$ ... *form*$_n$) destructures and compiles *arglist* and all the forms into pseudo-code, modifies *symbol*'s function definition cell to point to the resulting pseudo-code, thereby making *symbol* a macro. DEFMACRO returns *symbol*.

Unlike DEFUN, DEFMACRO provides for the **destructuring** of *arglist*. This facility means that arguments are assigned to the symbols in *arglist* based on the linked list structure of *arglist*. If *arglist* is a symbol, the actual arguments in the macro call are assigned to the symbol. Otherwise the car of the arguments are recursively assigned to the car of *arglist* and the cdr of the arguments are assigned to the cdr of *arglist*.

The formal definition of DEFUN on the previous page is an example of the use of DEFMACRO's destructuring capability.

```
(PUTD DEFMACRO '(MACRO BODY
  (POP BODY)
  (LIST 'PUTD
        (LIST 'QUOTE (CAR BODY))
        (LIST 'QUOTE
              (LIST 'MACRO
                    'BODY
                    (LIST (LIST* 'LAMBDA
                                 (Leaves (CADR BODY)) (CDDR BODY))
                                 (Destructure (CADR BODY) '(CDR BODY))))))) )

(DEFUN Leaves (TREE)
  ((ATOM TREE)
    (LIST TREE) )
  ((NULL (CDR TREE))
    (Leaves (CAR TREE)) )
  (NCONC (Leaves (CAR TREE)) (Leaves (CDR TREE))) )

(DEFUN Destructure (TREE FORM)
  ((ATOM TREE)
    (LIST FORM) )
  ((NULL (CDR TREE))
    (Destructure (CAR TREE) (LIST 'CAR FORM)) )
  (NCONC (Destructure (CAR TREE) (LIST 'CAR FORM))
         (Destructure (CDR TREE) (LIST 'CDR FORM))) )
```

CONDENSE: NIL Control variable

As a function definition is being compiled, a search is made to find existing pseudo-code equivalent to each subexpression in the definition. If such equivalent pseudo-code is found, it is used instead of creating more pseudo-code, thereby saving memory. This process is called **condensing** pseudo-code.

If *CONDENSE* is NIL, the condensing search is limited to the definition's own pseudo-code. Although it increases the pseudo-code space memory usage, limiting the condensing search space greatly reduces the time required to load large source files.

If *CONDENSE* is nonNIL, the condensing search spans the entire existing pseudo-code. This **full condensing** reduces the amount of memory required to store a given set of function definitions at a cost of increased time required to load large source files. Using a small set of generic names (e.g. NUM1, NUM2, LST, OBJ, ATM, etc.) in the formal argument lists of user defined functions can significantly increase the code density benefits derived from full condensing.

When a user-defined function is redefined using PUTD, MOVD, or REMD, the pseudo-code used to store the definition is automatically reclaimed if full condensing has *not* been performed (i.e. if *CONDENSE* has been NIL) since the function was defined. Thus, it is wise to have *CONDENSE* NIL during interactive program development when functions are frequently redefined.

5.4 Numbers

The functions in this section implement the basic mathematical operations for both integer and rational numbers. The user can select either exact or approximate **rational arithmetic**. For exact rational arithmetic the size of integers, numerators, and denominators is "limited" to thousands of digits. For approximate rational (also called **floating slash**) arithmetic the precision used can be adjusted to any desired degree of accuracy within this limitation.

If a function requiring integer arguments is called with a noninteger argument, a Noninteger Argument error break occurs. If a function requiring numeric (i.e. integer or rational) arguments is called with a nonnumeric argument, a Nonnumeric Argument error break occurs. If a function is called with too few arguments, an Insufficient Arguments error break occurs. If an attempt is made to divide by zero, a Zero Divide error break occurs.

When an **error break** occurs, a diagnostic message, the errant function call, and the break option prompt are displayed on the console. The break allows the user to determine the cause of the error and how to proceed (see Section 4.3).

The control variables *READ-BASE*, *PRINT-BASE*, and *PRINT-POINT* affect how numbers are expressed for input and output (see Section 5.8). Numerical results in the examples in this section are shown as they would appear with these variables set to their default value.

Numbers can be entered using either **decimal notation** (e.g. -0.75) or **rational notation** (e.g. -3/4). Note that there should be no spaces around a dot, slash, or minus sign. Also at least one decimal digit should precede a dot.

Regardless of how a number is entered, internally it is stored as the reduced ratio of two integers. This eliminates the many problems that can result from having a nonunique means of storing numbers. For example, in **muLISP** the form

```
(= -0.75 -3/4 -6/8)
```

returns T, as it should.

5.4.1 Precision

The control variable in this subsection selects either exact or approximate mode rational arithmetic. The functions determine the precision and underflow threshold used for numerical operations.

The rounding algorithm used for the approximate rational arithmetic was inspired by the work of Matula and Kornerup (see D. W. Matula and P. Kornerup, *Finite Precision Rational Arithmetic: Slash Number Systems*, **IEEE Transactions of Computers**, C-34, No. 1, January, 1985, pp. 3-18).

PRECISION [*n*] — Function
EXACT-MODE: NIL — Control variable

If *n* is a positive integer, (PRECISION *n*) sets the precision to *n* and returns the previous precision. If *n* is omitted or not a positive integer, (PRECISION *n*) returns the current precision without changing it. The initial precision setting is 1.

If *EXACT-MODE* is nonNIL, the rational numerical functions (+, -, *, /, GCD, and LCM) use exact rational arithmetic and do *not* round results.

If *EXACT-MODE* is NIL and the current precision is *n*, the numerical functions use approximate rational arithmetic to round results as follows: a) if the exact result is a ratio having a magnitude greater than 65536^n, the result is rounded to the truncated integer quotient of its numerator over its denominator; b) if the exact result is a ratio having a magnitude less than 65536^{-n}, the result is rounded to the reciprocal of the truncated integer quotient of its denominator over its numerator; otherwise c) if the exact result is a ratio requiring a total of more than $n+1$ words of memory to store its numerator and denominator, the result is rounded to one of the two closest rational approximations that require a total of no more than $n+1$ words of memory to store its numerator and denominator. (A word is equivalent to 2 bytes or 16 bits of memory).

In short, a precision of *n* generally provides about $2.5 + 4.5n$ decimal digits of accuracy. Thus, a precision of 1 roughly corresponds to **single precision** floating point; a precision of 3 roughly corresponds to **double precision** floating point; and so on.

```
(DEFUN PRECISION (N)
  ((AND (INTEGERP N) (PLUSP N))
    (set precision to N)
    (return previous precision) )
  (return current precision) )
```

```
$ 1.00001                                                    ;Does round to 1
1

$ (PRECISION 3)                                          ;Set double precision
1

$ 1.00001                                                ;Does not round to 1
1.00001

$ (SETQ *EXACT-MODE* T)                                   ;Set for exact mode
T

$ (SETQ *PRINT-POINT* 30)                                   ;Print 30 digits
30

$ 1.000000000000000000000000000001                       ;Does not round to 1
1.000000000000000000000000000001

$ (SETQ *EXACT-MODE* NIL)                             ;Set for approximate mode
NIL

$ (PRECISION 1)                                       ;Restore single precision
3
```

UNDERFLOW [*n*] Function

If *n* is a positive integer, (UNDERFLOW *n*) sets the value at which underflow occurs to 65536^{-n} and returns the previous underflow setting. If *n* is zero, (UNDERFLOW *n*) prevents underflow from occurring and returns the previous underflow setting. If *n* is omitted or is neither zero nor a positive integer, (UNDERFLOW *n*) returns the current setting of underflow without changing it. The initial underflow setting is 0.

Sometimes when doing calculations with approximate arithmetic, it is desirable to have numbers smaller in absolute value than a specified number "underflow" to 0. For example, when computing a series approximation of a function, the series can be terminated when the terms of the series underflow to zero.

The following table gives the approximate numerical value at which underflow occurs for various settings of *n*:

n	Underflow	n	Underflow	n	Underflow	n	Underflow
1	$1.53\ 10^{-5}$	4	$5.42\ 10^{-20}$	7	$1.93\ 10^{-34}$	10	$6.84\ 10^{-49}$
2	$2.33\ 10^{-10}$	5	$8.27\ 10^{-25}$	8	$2.94\ 10^{-39}$	11	$1.04\ 10^{-53}$
3	$3.55\ 10^{-15}$	6	$1.26\ 10^{-29}$	9	$4.48\ 10^{-44}$	12	$1.59\ 10^{-58}$

```
(DEFUN UNDERFLOW (N)
  ((ZEROP N)
    (disable underflow feature)
    (return previous underflow setting) )
  ((AND (INTEGERP N) (PLUSP N))
    (set underflow point to 65536^-N)
    (return previous underflow setting) )
  (return current underflow setting) )

$ 0.000001                                              ;No underflow
0.000001

$ (UNDERFLOW 1)                                         ;Set underflow to 1
0                                                  ;Returns previous setting

$ 0.000001                                              ;Underflows to 0
0

$ (UNDERFLOW 0)                                         ;Disable underflow
7

$ 0.00000000000000000000000000001                       ;No underflow
0.00000000000000000000000000001
```

5.4.2 Numerical Predicates

The numerical predicates are divided into recognizers and comparators. Since the numerical recognizers are used to identify objects, they do *not* generate a Nonnumeric Argument error break, even if given a symbol or a cons as an argument; they simply return NIL. However the numerical comparators, used to compare two or more numbers, do generate a Nonnumeric Argument error break if any argument is *not* a number.

ZEROP [*object*] Function
PLUSP [*object*] Function
MINUSP [*object*] Function

If *object* is the number zero, (ZEROP *object*) returns T, otherwise it returns NIL.

If *object* is a positive number, (PLUSP *object*) returns T, otherwise it returns NIL.

If *object* is a negative number, (MINUSP *object*) returns T, otherwise it returns NIL.

```
(DEFUN ZEROP (OBJ)
  (AND (NUMBERP OBJ) (EQL OBJ 0)) )

(DEFUN PLUSP (OBJ)
  (AND (NUMBERP OBJ) (> OBJ 0)) )

(DEFUN MINUSP (OBJ)
  (AND (NUMBERP OBJ) (< OBJ 0)) )

$ (ZEROP 0.0)
T

$ (ZEROP 3.1416)
NIL

$ (ZEROP 'DOG)
NIL

$ (PLUSP 3.1416)
T

$ (PLUSP 0)
NIL

$ (MINUSP 3.1416)
NIL

$ (MINUSP -7/3)
T
```

ODDP [*object*] Function
EVENP [*object*] Function

If *object* is an odd integer, (ODDP *object*) returns T, otherwise it returns NIL.

If *object* is an even integer, (EVENP *object*) returns T, otherwise it returns NIL.

```
(DEFUN ODDP (OBJ)
  (AND (INTEGERP OBJ) (NOT (ZEROP (REM OBJ 2)))) )

(DEFUN EVENP (OBJ)
  (AND (INTEGERP OBJ) (ZEROP (REM OBJ 2))) )

$ (ODDP 12)
NIL

$ (ODDP -41)
T

$ (ODDP 3.1416)
NIL

$ (ODDP 'DOG)
NIL

$ (EVENP 12)
T

$ (EVENP -41)
NIL

$ (EVENP 3.1416)
NIL

$ (EVENP 'DOG)
NIL
```

$= [n_1, n_2, ..., n_m]$ Function
$/= [n_1, n_2, ..., n_m]$ Function

If n_1 is equal to n_2, and n_2 is equal to n_3, and ..., and n_{m-1} is equal to n_m, then $(= n_1 n_2 ... n_m)$ returns T, otherwise it returns NIL. (=) and $(= n)$ return T.

If n_1 is *not* equal to n_2, and n_2 is *not* equal to n_3, and ..., and n_{m-1} is *not* equal to n_m, then $(/= n_1 n_2 ... n_m)$ returns T, otherwise it returns NIL. (/=) and $(/= n)$ return T.

Note that if /= returns T, no *adjacent* pair of arguments are equal. However, it can still be the case that two *nonadjacent* arguments are equal.

```
(DEFUN = LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((EQL (CAR LST) (CADR LST))
      (APPLY '= (CDR LST)) ) )
  ((BREAK (LIST '= (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '= (CDR LST)) ) )

(DEFUN /= LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((EQL (CAR LST) (CADR LST)) NIL)
    (APPLY '/= (CDR LST)) )
  ((BREAK (LIST '/= (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '/= (CDR LST)) ) )

$ (= 4 4 7)
NIL

$ (= 3 3.0)
T

$ (= 0.75 6/8)
T

$ (/= 4 4 7)
NIL

$ (/= 3 3.0)
NIL

$ (/= 0.75 6/8)
NIL

$ (/= 6 -2 6)                                    ;No adjacent pairs are equal
T
```

< $[n_1, n_2, ..., n_m]$ Function
> $[n_1, n_2, ..., n_m]$ Function

If $n_1 < n_2$, and $n_2 < n_3$, and ..., and $n_{m-1} < n_m$, then (< n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (<) and (< n) return T.

If $n_1 > n_2$, and $n_2 > n_3$, and ..., and $n_{m-1} > n_m$, then (> n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (>) and (> n) return T.

The comparator < is commonly used to determine if a number lies within an **open interval** by calling it with three arguments: the number and the interval's end-points. For example, the test

```
(< 10 NUM 100)
```

determines if NUM is between 10 and 100.

```
(DEFUN < LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((less than (CAR LST) (CADR LST))
      (APPLY '< (CDR LST)) ) )
  ((BREAK (LIST '< (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '< (CDR LST)) ) )

(DEFUN > LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((greater than (CAR LST) (CADR LST))
      (APPLY '> (CDR LST)) ) )
  ((BREAK (LIST '> (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '> (CDR LST)) ) )

$ (< 5 9)
T

$ (< 4 -7)
NIL

$ (< 3/5 2/3)
T

$ (> 5 9)
NIL

$ (> 7 3 4)
NIL

$ (> 7/2 3.2)
T
```

< = $[n_1, n_2, ..., n_m]$ Function
> = $[n_1, n_2, ..., n_m]$ Function

If $n_1 \leq n_2$, and $n_2 \leq n_3$, and ..., and $n_{m-1} \leq n_m$, then (< = n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (< =) and (< = n) return T.

If $n_1 \geq n_2$, and $n_2 \geq n_3$, and ..., and $n_{m-1} \geq n_m$, then (> = n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (> =) and (> = n) return T.

The comparator < = is commonly used to determine if a number lies in a **closed interval** by calling it with three arguments: the number and the interval's end-points. For example, the test

```
(<= 10 NUM 100)
```

determines if NUM is between 10 and 100 inclusive.

```
(DEFUN <= LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((> (CAR LST) (CADR LST))  NIL)
    (APPLY '<= (CDR LST)) )
  ((BREAK (LIST '<= (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '<= (CDR LST)) ) )

(DEFUN >= LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((< (CAR LST) (CADR LST))  NIL)
    (APPLY '>= (CDR LST)) )
  ((BREAK (LIST '>= (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '>= (CDR LST)) ) )

$ (<= 5 9 9)
T

$ (<= 4 -7)
NIL

$ (<= 3 3.0)
T

$ (>= 4 -7)
T

$ (>= 3 3.0)
T

$ (>= 3/5 2/3)
NIL
```

MAX [n_1, n_2, ..., n_m] Function
MIN [n_1, n_2, ..., n_m] Function

(MAX n_1 n_2 ... n_m) returns its greatest argument (i.e. the argument furthest to the right on the real number line).

(MIN n_1 n_2 ... n_m) returns its least argument (i.e. the argument furthest to the left on the real number line).

If these functions are called with no arguments, an Insufficient Arguments error break occurs.

```
(DEFUN MAX LST
  ((NULL LST)
    (BREAK (LIST 'MAX) "Insufficient Arguments") )
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((< (CAR LST) (CADR LST))
      (APPLY 'MAX (CADR LST) (CDDR LST)) )
    (APPLY 'MAX (CAR LST) (CDDR LST)) )
  (APPLY 'MAX (BREAK (LIST 'MAX (CAR LST) (CADR LST)) "Nonnumeric Argument")
              (CDDR LST)) )

(DEFUN MIN LST
  ((NULL LST)
    (BREAK (LIST 'MIN) "Insufficient Arguments") )
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((< (CAR LST) (CADR LST))
      (APPLY 'MIN (CAR LST) (CDDR LST)) )
    (APPLY 'MIN (CADR LST) (CDDR LST)) )
  (APPLY 'MIN (BREAK (LIST 'MIN (CAR LST) (CADR LST)) "Nonnumeric Argument")
              (CDDR LST)) )

$ (MAX 5 -7 4)
5

$ (MAX 2/3 0.6)
0.6666666

$ (MAX -4)
-4

$ (MIN 2/3 0.6)
0.6
```

5.4.3 Arithmetic Operators

The functions in this subsection perform the basic mathematical operations on both integer and rational numbers. They generate a Nonnumeric Argument error break if any argument is *not* a number.

+ $[n_1, n_2, ..., n_m]$ Function
− $[n_1, n_2, ..., n_m]$ Function

(+ n_1 n_2 ... n_m) returns the sum of the numbers n_1 through n_m. (+) returns 0, the identity for addition.

(− n_1 n_2 ... n_m) returns the difference between n_1 and the sum of n_2 through n_m. (− n) returns minus n. (−) generates an Insufficient Arguments error break.

```
(DEFUN + LST
  ((NULL LST)  0)
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY '+ ((CAR LST) plus (CADR LST)) (CDDR LST)) )
  (APPLY '+ (+TRAP (CAR LST) (CADR LST)) (CDDR LST)) )

(DEFUN - LST
  ((NULL LST)
    (BREAK (LIST '-) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (minus (CAR LST)) )
    (-TRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((NULL (CDDR LST))
      ((CAR LST) minus (CADR LST)) )
    (APPLY '- ((CAR LST) minus (CADR LST)) (CDDR LST)) )
  (APPLY '- (-TRAP (CAR LST) (CADR LST)) (CDDR LST)) )

$ (+ 1/6 0.7)
0.8666666

$ (- 12 5)
7

$ (- 8)
-8
```

* $[n_1, n_2, ..., n_m]$ Function
/ $[n_1, n_2, ..., n_m]$ Function

(* n_1 n_2 ... n_m) returns the product of the numbers n_1 through n_m. (*) returns 1, the identity for multiplication.

(/ n_1 n_2 ... n_m) returns the result of n_1 divided by the product of n_2 through n_m. (/ n) returns the reciprocal of n. (/) generates an Insufficient Arguments error break. If any argument other than the first is zero, a Zero Divide error break occurs.

```
(DEFUN * LST
  ((NULL LST)  1)
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY '* ((CAR LST) times (CADR LST)) (CDDR LST)) )
  (APPLY '* (*TRAP (CAR LST) (CADR LST)) (CDDR LST)) )

(DEFUN / LST
  ((NULL LST)
    (BREAK (LIST '/) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      ((ZEROP (CAR LST))
        (/TRAP (CAR LST)) )
      (reciprocal of (CAR LST)) )
    (/TRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((NULL (CDDR LST))
      ((ZEROP (CADR LST))
        (/TRAP (CAR LST) (CADR LST)) )
      (quotient (CAR LST) (CADR LST)) )
    ((ZEROP (CADR LST))
      (APPLY '/ (/TRAP (CAR LST) (CADR LST)) (CDDR LST)) )
    (APPLY '/ (quotient (CAR LST) (CADR LST)) (CDDR LST)) )
  (APPLY '/ (/TRAP (CAR LST) (CADR LST)) (CDDR LST)) )

$ (* 3 4 5)
60

$ (* 5/6 -0.7)
-0.5833333

$ (/ 12 8)
1.5

$ (/ 5)
0.2
```

ADD1 [*n*] Function
SUB1 [*n*] Function

(ADD1 *n*) returns (+ *n* 1). (ADD1) generates an Insufficient Arguments error break.

(SUB1 *n*) returns (– *n* 1). (SUB1) generates an Insufficient Arguments error break.

```
(DEFUN ADD1 (N)
  ((NUMBERP N)
    (+ N 1) )
  (BREAK (LIST 'ADD1 N) "Nonnumeric Argument") )

(DEFUN SUB1 (N)
  ((NUMBERP N)
    (- N 1) )
  (BREAK (LIST 'SUB1 N) "Nonnumeric Argument") )

$ (ADD1 3)
4

$ (ADD1 2/3)
1.6666666

$ (ADD1 -3)
-2

$ (SUB1 3)
2

$ (SUB1 0.25)
-0.75

$ (SUB1 -3)
-4
```

INCQ [*symbol*, *n*] Special form
DECQ [*symbol*, *n*] Special form

If *symbol* is a symbol and *n* is a number, (INCQ *symbol n*) adds *n* to the value of *symbol*, sets *symbol* to the result, and returns the result. If *n* is omitted, *symbol* is incremented by 1.

If *symbol* is a symbol and *n* is a number, (DECQ *symbol n*) subtracts *n* from the value of *symbol*, sets *symbol* to the result, and returns the result. If *n* is omitted, *symbol* is decremented by 1.

If the first argument of INCQ or DECQ is not a symbol, a Nonsymbolic Argument error break occurs.

```
(DEFMACRO INCQ (SYM N)
  ((NULL N)
    (LIST 'SETQ SYM (LIST 'ADD1 SYM)) )
  (LIST 'SETQ SYM (LIST '+ SYM N)) )

(DEFMACRO DECQ (SYM N)
  ((NULL N)
    (LIST 'SETQ SYM (LIST 'SUB1 SYM)) )
  (LIST 'SETQ SYM (LIST '- SYM N)) )

$ (SETQ CTR 5)                                    ;Set counter to 5
5

$ (INCQ CTR)                                      ;Increment counter
6

$ (INCQ CTR 2)                                    ;Add 2 to counter
8

$ (DECQ CTR 5)                                 ;Subtract 5 from counter
3

$ CTR                                             ;Value of counter
3
```

INCF [*place*, *n*] Library macro
DECF [*place*, *n*] Library macro

If *place* is a valid place form and *n* is a number, (INCF *place n*) adds *n* to the value in *place*, stores the result in *place*, and returns the result. If *n* is omitted, *place* is incremented by 1.

If *place* is a valid place form and *n* is a number, (DECF *place n*) subtracts *n* from the value in *place*, stores the result in *place*, and returns the result. If *n* is omitted, *place* is decremented by 1.

If the first argument of INCF or DECF is not a valid place form (see SETF in Section 5.3.1), an Invalid Place Form error break occurs. INCF and DECF are defined in COMMON.LSP.

```
(DEFMACRO INCF (PLACE N)
  ((NULL N)
    (MAKE-SET-FORM PLACE (LIST 'ADD1 PLACE)) )
  (MAKE-SET-FORM PLACE (LIST '+ PLACE N)) )

(DEFMACRO DECF (PLACE N)
  ((NULL N)
    (MAKE-SET-FORM PLACE (LIST 'SUB1 PLACE)) )
  (MAKE-SET-FORM PLACE (LIST '- PLACE N)) )

$ (SETQ LST '(5 2 3))                                    ;Initialize LST
(5 2 3)

$ (INCF (CAR LST))                                       ;Increment car of LST
6

$ (INCF (NTH 1 LST) 2)                                   ;Add 2 to second of LST
4

$ (DECF (CADR LST) 5)                                    ;Subtract 5 from second of LST
-1

$ LST                                                    ;Value of LST
(6 -1 3)
```

GCD [n_1, n_2, ..., n_m] Function

(GCD n_1 n_2 ... n_m) returns the gcd (**greatest common divisor**) of the numbers n_1 through n_m. (GCD n) returns the absolute value of n. (GCD) returns 0, the identity for this operation.

The gcd of two integers is the largest non-negative integer that evenly divides both integers. The gcd of two rational numbers is the gcd of the numerators divided by the lcm of the denominators. In either case if the gcd is divided into the two numbers, the result is the smallest two integers having the same ratio as the original two numbers.

```
(DEFUN GCD LST
  ((NULL LST)  0)
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (ABS (CAR LST)) )
    (GCDTRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY 'GCD (/ (Integer-GCD (NUMERATOR (CAR LST)) (NUMERATOR (CADR LST)))
                   (Integer-LCM (DENOMINATOR (CAR LST)) (DENOMINATOR (CADR LST))))
                (CDDR LST)) )
  (APPLY 'GCD (GCDTRAP (CAR LST) (CADR LST)) (CDDR LST)) )

(DEFUN Integer-GCD (N M)
  ((ZEROP M)
    (ABS N) )
  (Integer-GCD M (REM N M)) )

(DEFUN Integer-LCM (N M)
  (/ (* N M) (Integer-GCD N M)) )

$ (GCD 4 6)
2

$ (GCD 12 -16 20)
4

$ (GCD 14/15 8/9)
0.0444444

$ (GCD 0 5)
5

$ (GCD 0 0)
0

$ (GCD -7)
7
```

LCM [n_1, n_2, ..., n_m] Function

(LCM n_1 n_2 ... n_m) returns the lcm (**least common multiple**) of the numbers n_1 through n_m. (LCM n) returns the absolute value of n. (LCM) generates an Insufficient Arguments error break, since this operation has no finite identity.

The lcm of two integers is the least non-negative integer that is a multiple of both integers. In general, the lcm of the two numbers n and m, either integer or rational, can be defined as

$$\frac{|\, n \;\; m \,|}{\gcd\,(n, m)}$$

```
(DEFUN LCM LST
  ((NULL LST)
    (BREAK (LIST 'LCM) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (ABS (CAR LST)) )
    (LCMTRAP (CAR LST)) )
  ((OR (ZEROP (CAR LST)) (ZEROP (CADR LST)))  0)
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY 'LCM (/ (ABS (* (CAR LST) (CADR LST))) (GCD (CAR LST) (CADR LST)))
                (CDDR LST)) )
  (APPLY 'LCM (LCMTRAP (CAR LST) (CADR LST)) (CDDR LST)) )

$ (LCM 4 6)
12

$ (LCM 12 -16 20)
240

$ (LCM 14/15 8/9)
18.6666666                                        ;This result is = to 56/3

$ (LCM 0 5)
0
```

5.4.4 Component Extraction

ABS [*n*] Function
SIGNUM [*n*] Function

If *n* is a number, (ABS *n*) returns the **absolute value** of *n*. If *n* is not a number, ABSTRAP is called to generate a Nonnumeric Argument error break (see Section 5.10.3).

If *n* is a number, (SIGNUM *n*) returns 1, -1, or 0 depending on whether *n* is positive, negative, or zero. If *n* is not a number, a Nonnumeric Argument error break occurs.

```
(DEFUN ABS (N)
  ((NUMBERP N)
    ((MINUSP N)
      (- N) )
    N )
  (ABSTRAP N) )

(DEFUN SIGNUM (N)
  ((NUMBERP N)
    ((ZEROP N) 0)
    ((PLUSP N) 1)
    -1 )
  (BREAK (LIST 'SIGNUM N) "Nonnumeric Argument") )

$ (ABS 4/6)
0.6666666

$ (ABS -3)
3

$ (ABS 0)
0

$ (SIGNUM 7/2)
1

$ (SIGNUM 0)
0

$ (SIGNUM -0.2)
-1
```

NUMERATOR [*n*] Function
DENOMINATOR [*n*] Function

If *n* is a ratio, (NUMERATOR *n*) returns the numerator of *n*. If *n* is an integer, (NUMERATOR *n*) returns *n*. The numerator is always an integer. If *n* is not a number, NUMTRAP is called to generate a Nonnumeric Argument error break (see Section 5.10.3).

If *n* is a ratio, (DENOMINATOR *n*) returns the denominator of *n*. If *n* is an integer, (DENOMINATOR *n*) returns 1. The denominator is always a positive integer. If *n* is not a number, DENTRAP is called to generate a Nonnumeric Argument error break (see Section 5.10.3).

```
(DEFUN NUMERATOR (N)
  ((NUMBERP N)
    ((INTEGERP N)  N)
    (the numerator of N) )
  (NUMTRAP N) )

(DEFUN DENOMINATOR (N)
  ((NUMBERP N)
    ((INTEGERP N)  1)
    (the denominator of N) )
  (DENTRAP N) )

$ (NUMERATOR 10/8)
5

$ (DENOMINATOR 10/8)
4

$ (NUMERATOR 0.75)
3

$ (DENOMINATOR 0.75)
4

$ (NUMERATOR -8)
-8

$ (DENOMINATOR -8)
1
```

FLOOR [*n*, *m*] Function
CEILING [*n*, *m*] Function

If *n* is an integer, (FLOOR *n*) returns *n*. If *n* is a ratio, FLOOR returns the largest integer less than *n*. In other words, FLOOR truncates *n* toward minus infinity. If *n* and *m* are numbers, (FLOOR *n m*) truncates toward minus infinity the quotient of *n* divided by *m*. (FLOOR *n m*) equals (FLOOR (/ *n m*)), but is more efficient.

If *n* is an integer, (CEILING *n*) returns *n*. If *n* is a ratio, CEILING returns the smallest integer greater than *n*. In other words, CEILING truncates *n* toward plus infinity. If *n* and *m* are numbers, (CEILING *n m*) truncates toward plus infinity the quotient of *n* divided by *m*. (CEILING *n m*) equals (CEILING (/ *n m*)), but is more efficient.

If the second argument of FLOOR or CEILING (i.e. the divisor) is zero, a Zero Divide error break occurs.

```
(DEFUN FLOOR (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'FLOOR N M) "Zero Divide") )
    (FLOOR (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N)  N)
    (the largest integer less than N) )
  (BREAK (LIST 'FLOOR N M) "Nonnumeric Argument") )

(DEFUN CEILING (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'CEILING N M) "Zero Divide") )
    (CEILING (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N)  N)
    (the smallest integer greater than N) )
  (BREAK (LIST 'CEILING N M) "Nonnumeric Argument") )
```

```
$ (FLOOR 7.3)
7

$ (FLOOR -3.5)
-4

$ (FLOOR 8 3)
2

$ (FLOOR -15/4 2)
-2
```

```
$ (CEILING 7.3)
8

$ (CEILING -3.5)
-3

$ (CEILING 8 3)
3

$ (CEILING -15/4 2)
-1
```

TRUNCATE [*n*, *m*] Function
ROUND [*n*, *m*] Function

If *n* is an integer, (TRUNCATE *n*) returns *n*. If *n* is a positive ratio, TRUNCATE returns the largest integer less than *n*. If *n* is a negative ratio, TRUNCATE returns the smallest integer greater than *n*. In other words, TRUNCATE truncates *n* toward zero. If *n* and *m* are numbers, (TRUNCATE *n m*) truncates toward zero the quotient of *n* divided by *m*. (TRUNCATE *n m*) equals (TRUNCATE (/ *n m*)), but is more efficient.

If *n* is an integer, (ROUND *n*) returns *n*. If *n* is a ratio, ROUND returns the integer closest in value to *n*. If *n* is equidistant between two integers (i.e. 2 *n* is an integer), ROUND returns the one that is divisible by two. If *n* and *m* are numbers, (ROUND *n m*) rounds the quotient of *n* divided by *m*. (ROUND *n m*) equals (ROUND (/ *n m*)), but is more efficient.

If the second argument of TRUNCATE or ROUND (i.e. the divisor) is zero, a Zero Divide error break occurs.

```
(DEFUN TRUNCATE (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'TRUNCATE N M) "Zero Divide") )
    (TRUNCATE (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N)  N)
    ((PLUSP N)  (the largest integer less than N) )
    (the smallest integer greater than N) )
  (BREAK (LIST 'TRUNCATE N M) "Nonnumeric Argument") )

(DEFUN ROUND (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'ROUND N M) "Zero Divide") )
    (ROUND (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N)  N)
    ((INTEGERP (* 2 N))
      ((EVENP (TRUNCATE N))  (TRUNCATE N) )
      (ADD1 (TRUNCATE N)) )
    (the integer closest to N) )
  (BREAK (LIST 'ROUND N M) "Nonnumeric Argument") )
```

```
$ (TRUNCATE -2.4)
-3

$ (TRUNCATE 7 2)
3
```

```
$ (ROUND -2.4)
-2

$ (ROUND 7 2)
4
```

MOD $[n, m]$ Function
REM $[n, m]$ Function

If n and m are numbers, (MOD n m) returns the **modulus** of n divided by m. m defaults to 1. MOD is defined so that FLOOR and MOD preserve the fundamental relation between quotients and remainders:

$$n = m \text{ floor } (n, m) + \text{mod } (n, m)$$

If n and m are numbers, (REM n m) returns the **remainder** of n divided by m. m defaults to 1. REM is defined so that TRUNCATE and REM preserve the fundamental relation between quotients and remainders:

$$n = m \text{ truncate } (n, m) + \text{rem } (n, m)$$

If the second argument of MOD or REM (i.e. the divisor) is zero, a Zero Divide error break occurs.

```
(DEFUN MOD (N M)
  ((NULL M)
    (MOD N 1) )
  ((AND (NUMBERP N) (NUMBERP M))
    ((ZEROP M)
      (BREAK (LIST 'MOD N M) "Zero Divide") )
    (- N (* M (FLOOR N M))) )
  (BREAK (LIST 'MOD N M) "Nonnumeric Argument") )

(DEFUN REM (N M)
  ((NULL M)
    (REM N 1) )
  ((AND (NUMBERP N) (NUMBERP M))
    ((ZEROP M)
      (BREAK (LIST 'REM N M) "Zero Divide") )
    (- N (* M (TRUNCATE N M))) )
  (BREAK (LIST 'REM N M) "Nonnumeric Argument") )
```

```
$ (MOD 7 3)
1

$ (MOD -7 3)
2

$ (MOD 7 -3)
-2

$ (MOD -7 -3)
-1
```

```
$ (REM 7 3)
1

$ (REM -7 3)
-1

$ (REM 7 -3)
1

$ (REM -7 -3)
-1
```

DIVIDE [*n*, *m*] Function

If *n* is a number, (DIVIDE *n*) returns a cons whose car is (TRUNCATE *n*) and whose cdr is (REM *n*). If *n* and *m* are numbers, (DIVIDE *n m*) returns a cons whose car is (TRUNCATE *n m*) and whose cdr is (REM *n m*).

If the second argument of DIVIDE (i.e. the divisor) is zero, a Zero Divide error break occurs.

When both a truncated quotient and remainder are required, using the DIVIDE function is more efficient than computing them separately.

```
(DEFUN DIVIDE (N M)
  ((AND (NUMBERP N) (NUMBERP M))
    ((ZEROP M)
      (BREAK (LIST 'DIVIDE N M) "Zero Divide") )
    (CONS (TRUNCATE N M) (REM N M)) )
  (BREAK (LIST 'DIVIDE N M) "Nonnumeric Argument") )

$ (DIVIDE 7 3)
(2 . 1)

$ (DIVIDE -7 3)
(-2 . -1)

$ (DIVIDE 7 -3)
(-2 . 1)

$ (DIVIDE -7 -3)
(2 . -1)
```

5.4.5 Bit-wise Logical Operators

The functions in this subsection are for performing bit-wise logical operations over the integers. If called with noninteger arguments, a Noninteger Argument error break occurs.

LOGNOT [*n*] — Function
LOGTEST [*n*, *m*] — Library function
LOGBITP [*index*, *n*] — Library function

(LOGNOT *n*) returns the bit-wise **logical not** of *n* (i.e. the integer whose bits are the complement of the bits in *n*). Since LOGNOT operates as if negative integers are stored in "half-infinite" two's-complement notation, the result is always of the opposite sign as *n*.

If any 1 bits in *n* are also 1 bits in *m*, (LOGTEST *n m*) returns T, otherwise it returns NIL. LOGTEST is defined in COMMON.LSP.

If the bit in *n* whose weight is 2^{index} is 1, (LOGBITP *index n*) returns T, otherwise it returns NIL. LOGBITP is defined in COMMON.LSP.

```
$ (LOGNOT 9)                                    ;Complement 9
-10

$ (LOGNOT -1)                                   ;Complement -1
0

$ (LOGTEST 4 13)                          ;Test the 2nd bit in 13
T

$ (LOGBITP 2 13)                          ;Test the 2nd bit in 13
T
```

LOGIOR [n_1, n_2, ..., n_m] Function
LOGXOR [n_1, n_2, ..., n_m] Function
LOGAND [n_1, n_2, ..., n_m] Function
LOGEQV [n_1, n_2, ..., n_m] Library function

These functions perform logical operations on the bits of their integer arguments. They operate as if integers are stored in "half-infinite" two's-complement notation.

(LOGIOR n_1 n_2 ... n_m) returns the bit-wise **logical inclusive or** of its arguments. (LOGIOR) returns 0, the identity for this operation.

(LOGXOR n_1 n_2 ... n_m) returns the bit-wise **logical exclusive or** of its arguments. (LOGXOR) returns 0, the identity for this operation.

(LOGAND n_1 n_2 ... n_m) returns the bit-wise **logical and** of its arguments. (LOGAND) returns -1, the identity for this operation.

(LOGEQV n_1 n_2 ... n_m) returns the bit-wise **logical equivalence** of its arguments. (LOGEQV) returns -1, the identity for this operation. LOGEQV is defined in COMMON.LSP.

```
(DEFUN LOGIOR LST
  (REDUCE '(LAMBDA (INT1 INT2)
      ((AND (INTEGERP INT1) (INTEGERP INT2))
        (the logical OR of INT1 and INT2) )
      (BREAK (LIST 'LOGIOR INT1 INT2) "Noninteger Argument") )
    LST 0) )

$ (LOGIOR 11 6)
15

$ (LOGXOR -6 6)
-4

$ (LOGAND 11 6)
2

$ (LOGEQV 11 6)
-14
```

SHIFT [*n*, *count*] Function
ASH [*n*, *count*] Library function

If *n* is an integer and if *count* is a positive integer, (SHIFT *n count*) returns the result of shifting *n* left by *count* bits. If *count* is a negative integer, *n* is shifted right *-count* bits. The result is always of the same sign as *n*. Shifting left (right) is a fast way to multiply (divide) a number by a power of 2.

ASH is similar to SHIFT except that it operates as if negative integers are stored in "half-infinite" two's-complement notation. ASH is defined in COMMON.LSP.

```
(DEFUN SHIFT (N COUNT)
  ((AND (INTEGERP N) (INTEGERP COUNT))
    ((ZEROP COUNT)  N)
    ((PLUSP COUNT)
      (SHIFT (* N 2) (SUB1 COUNT)) )
    (SHIFT (TRUNCATE N 2) (SUB1 COUNT)) )
  (BREAK (LIST 'SHIFT N COUNT) "Noninteger Argument") )

(DEFUN ASH (N COUNT)
  ((OR (>= N 0) (>= COUNT 0))
    (SHIFT N COUNT) )
  (ASH (SHIFT (SUB1 N) -1) (ADD1 COUNT)) )

$ (SHIFT 5 -1)                                          ;halve 5
2

$ (SHIFT 5 1)                                          ;double 5
10

$ (SHIFT 5 2)                                       ;quadruple 5
20

$ (SHIFT -7 -2)                                  ;logical right shift
-1

$ (ASH -7 -2)                                  ;arithmetic right shift
-2
```

5.4.6 Random Numbers

The functions in this subsection are for generating pseudo-random numbers. All are defined in COMMON.LSP.

RANDOM [*n*, *state*] Library function

If *n* is a positive integer, (RANDOM *n*) returns an approximately uniformly distributed random integer *x* such that $0 \le x < n$. If *n* is a number but not a positive integer, (RANDOM *n*) returns an approximately uniformly distributed random rational number *x* such that $0 \le x < |n|$, subject to the current precision as set by PRECISION (see Section 5.4.1).

The non-negative integer value of the random state variable *RANDOM-STATE* is used and maintained by RANDOM as the present state of the random-number generator. If *state* is zero or a positive integer, (RANDOM *n state*) sets *RANDOM-STATE* to *state* before returning (RANDOM *n*).

```
(DEFUN RANDOM (N STATE)
  (IF (RANDOM-STATE-P STATE)
      (SETQ *RANDOM-STATE* STATE) )
  ((AND (INTEGERP N) (PLUSP N))
    (LET ((TEMP 0)
          (COUNT 0))
      (LOOP
        (SETQ *RANDOM-STATE* (LOGAND 4294967295
                                     (ADD1 (* 3141592653 *RANDOM-STATE*))))
        (SETQ TEMP (+ (* TEMP 32) *RANDOM-STATE*))
        ((> (INCQ COUNT 32) (+ (INTEGER-LENGTH N) 10))
          (SHIFT (* N TEMP) (- COUNT)) ) ) ) )
  ((NUMBERP N)
    (LET ((TEMP (EXPT 2 (- (* 16 (ADD1 (PRECISION))) 10))))
      (/ (* (ABS N) (RANDOM TEMP)) TEMP) ) ) )

$ (RANDOM 100)                                  ;Random integer less than 100
84                                                        ;Result will vary

$ (RANDOM -100)                                 ;Random number less than 100
61.2102272                                                ;Result will vary
```

RANDOM-STATE Library control variable
MAKE-RANDOM-STATE [*state*] Library function

RANDOM-STATE is used to store the current state of the random number generator. The function RANDOM automatically updates the value of *RANDOM-STATE* when generating random numbers. *RANDOM-STATE* should only be set to values returned by the function MAKE-RANDOM-STATE. Initially *RANDOM-STATE* is set to (MAKE-RANDOM-STATE T).

If *state* is T, (MAKE-RANDOM-STATE *state*) returns a random-number state based on the system clock. If *state* is NIL or omitted, MAKE-RANDOM-STATE returns *RANDOM-STATE*. If *state* is an acceptable random-number state, MAKE-RANDOM-STATE returns *state*. Otherwise, MAKE-RANDOM-STATE returns NIL.

```
(DEFUN MAKE-RANDOM-STATE (STATE)
  ((NULL STATE)
    *RANDOM-STATE* )
  ((EQ STATE 'T)
    (TIME) )
  ((RANDOM-STATE-P STATE)
    STATE ) )

(SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 'T))

$ (SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 1000))
1000

$ (RANDOM 100)
45

$ (SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 1000))
1000

$ (RANDOM 100)                          ;Resetting *RANDOM-STATE* makes
45                                          ;RANDOM results repeatable
```

RANDOM-STATE-P [*object*] Library function

If *object* is an acceptable random-number state object, (RANDOM-STATE-P *object*) returns T, otherwise it returns NIL. The acceptable random-number states are integers greater than or equal to zero.

```
(DEFUN RANDOM-STATE-P (OBJ)
  (AND (INTEGERP OBJ) (>= OBJ 0)) )

$ (RANDOM-STATE-P 546)                    ;Positives are valid random states
T
$ (RANDOM-STATE-P (TIME))                 ;TIME gives valid random states
T
$ (RANDOM-STATE-P -73)                    ;Negatives are not valid random states
NIL
$ (RANDOM-STATE-P 12.7)                   ;Ratios are not valid random states
NIL
```

5.5 Print Names

The print name functions are for manipulating the print name strings associated with symbols and numbers. They are useful for text and language processing applications.

5.5.1 Character Functions

The functions in this subsection treat the first character of a symbol's print name string as a **character**. Functions are provided for recognizing, comparing, and converting the case of characters.

UPPER-CASE-P [*symbol*] Function
LOWER-CASE-P [*symbol*] Function

If the first character of *symbol*'s print name is an uppercase character, (UPPER-CASE-P *symbol*) returns T, otherwise it returns NIL. The **uppercase characters** are the letters *A* through *Z*.

If the first character of *symbol*'s print name is a lowercase character, (LOWER-CASE-P *symbol*) returns T, otherwise it returns NIL. The **lowercase characters** are the letters *a* through *z*.

```
(DEFUN UPPER-CASE-P (SYM)
  ((SYMBOLP SYM)
    (<= 65 (ASCII SYM) 90) ) )

(DEFUN LOWER-CASE-P (SYM)
  ((SYMBOLP SYM)
    (<= 97 (ASCII SYM) 122) ) )

$ (UPPER-CASE-P "D")
T

$ (UPPER-CASE-P "d")
NIL

$ (LOWER-CASE-P "D")
NIL

$ (LOWER-CASE-P "d")
T
```

ALPHA-CHAR-P [*symbol*] Function
NUMERIC-CHAR-P [*symbol*] Function
ALPHANUMERICP [*symbol*] Function

If the first character of *symbol*'s print name is an alphabetic character, (ALPHA-CHAR-P *symbol*) returns T, otherwise it returns NIL. The **alphabetic characters** are the letters *A* through *Z* and *a* through *z*.

If the first character of *symbol*'s print name is a numeric character, (NUMERIC-CHAR-P *symbol*) returns T, otherwise it returns NIL. The **numeric characters** are the digits *0* through *9*.

If the first character of *symbol*'s print name is an alphabetic or numeric character, (ALPHANUMERICP *symbol*) returns T, otherwise it returns NIL.

```
(DEFUN ALPHA-CHAR-P (SYM)
  (OR (UPPER-CASE-P SYM) (LOWER-CASE-P SYM)) )

(DEFUN NUMERIC-CHAR-P (SYM)
  ((SYMBOLP SYM)
    (<= 48 (ASCII SYM) 57) ) )

(DEFUN ALPHANUMERICP (SYM)
  (OR (ALPHA-CHAR-P SYM) (NUMERIC-CHAR-P SYM)) )

$ (ALPHA-CHAR-P "D")
T

$ (ALPHA-CHAR-P "3")
NIL

$ (NUMERIC-CHAR-P "D")
NIL

$ (NUMERIC-CHAR-P "3")
T

$ (ALPHANUMERICP "D")
T

$ (ALPHANUMERICP "3")
T

$ (ALPHANUMERICP "$")
NIL
```

DIGIT-CHAR-P [*symbol, radix*] Function

If *radix* is an integer between 2 and 36 inclusive and the first character of *symbol*'s print name is a character whose numeric weight is less than *radix*, (DIGIT-CHAR-P *symbol radix*) returns the numeric weight of the character. If the character is not less than *radix*, DIGIT-CHAR-P returns NIL. *radix* defaults to 10.

The **numeric weights** of the digits *0* through *9* are the integers 0 through 9. The weights of the letters *A* through *Z* (and also the letters *a* through *z*) are the integers 10 through 35.

```
(DEFUN DIGIT-CHAR-P (SYM RADIX)
  ((AND (INTEGERP RADIX) (<= 2 RADIX 36))
    ((NUMERIC-CHAR-P SYM)
      (SETQ SYM (- (ASCII SYM) 48))
      ((< SYM RADIX)  SYM) )
    ((ALPHA-CHAR-P SYM)
      (SETQ SYM (- (ASCII (CHAR-UPCASE SYM)) 55))
      ((< SYM RADIX)  SYM) ) )
  (DIGIT-CHAR-P SYM 10) )

$ (DIGIT-CHAR-P "5")                          ;5 is the weight of "5"
5

$ (DIGIT-CHAR-P "5" 2)                        ;"5" is not a binary digit
NIL

$ (DIGIT-CHAR-P "D")                          ;"D" is not a decimal digit
NIL

$ (DIGIT-CHAR-P "D" 16)                       ;13 is the weight of "D"
13
```

CHAR= [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR< [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR> [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR<= [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR>= [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR/= [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function

These predicates perform a **lexicographical comparison** on the first character of the print name of their arguments, taking into account the distinction between upper and lower case letters. If the comparison is true, they return T, otherwise they return NIL.

CHAR= performs an equal test, CHAR< performs a less than test, CHAR> performs a greater than test, CHAR<= performs a less than or equal test, CHAR>= performs a greater than or equal test, and CHAR/= performs a not equal test.

```
(DEFUN CHAR= (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (= (ASCII SYM1) (ASCII SYM2)) ) )

(DEFUN CHAR< (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (< (ASCII SYM1) (ASCII SYM2)) ) )

(DEFUN CHAR<= (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (<= (ASCII SYM1) (ASCII SYM2)) ) )

$ (CHAR= "A" "a")                              ;Case sensitive predicates
NIL

$ (CHAR= "peach" "plum" "pear")              ;Compares only the first character
T

$ (CHAR< "D" "C")                                  ;"D" ordered after "C"
NIL

$ (CHAR< "C" "D" "G")                               ;Characters in order
T

$ (CHAR< "D" "c")                          ;Uppercase ordered before lowercase
T
```

CHAR-EQUAL [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR-LESSP [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR-GREATERP [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR-NOT-GREATERP [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR-NOT-LESSP [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR-NOT-EQUAL [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function

These predicates perform a **lexicographical comparison** on the first character of the print name of their atomic arguments, making *no* distinction between upper and lower case letters. If the comparison is true, they return T, otherwise they return NIL.

CHAR-EQUAL performs an equal test, CHAR-LESSP performs a less than test, CHAR-GREATERP performs a greater than test, CHAR-NOT-GREATERP performs a less than or equal test, CHAR-NOT-LESSP performs a greater than or equal test, and CHAR-NOT-EQUAL performs a not equal test.

```
(DEFUN CHAR-EQUAL (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (= (ASCII (CHAR-UPCASE SYM1)) (ASCII (CHAR-UPCASE SYM2))) ) )

(DEFUN CHAR-LESSP (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (< (ASCII (CHAR-UPCASE SYM1)) (ASCII (CHAR-UPCASE SYM2))) ) )

(DEFUN CHAR-NOT-GREATERP (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (<= (ASCII (CHAR-UPCASE SYM1)) (ASCII (CHAR-UPCASE SYM2))) ) )

$ (CHAR-EQUAL "A" "a")                          ;Case insensitive predicates
T

$ (CHAR-EQUAL "peach" "plum" "pear")            ;Compares only the first character
T

$ (CHAR-LESSP "D" "C")                          ;"D" ordered after "C"
NIL

$ (CHAR-LESSP "C" "D" "G")                      ;Characters in order
T

$ (CHAR-LESSP "D" "c")                          ;Case ignored
NIL
```

ASCII [*atom*] Function
CHAR-CODE [*symbol*] Library function
CODE-CHAR [*n*] Library function

If *atom* is a symbol, (ASCII *atom*) returns the **ASCII character code** for the first character of *atom*'s print name. If *atom* is an integer between 0 and 255 inclusive, (ASCII *atom*) returns the symbol whose print name is a single character which has an ASCII code equal to *atom*. Otherwise ASCII returns NIL.

ASCII is useful for generating the **control characters** necessary to perform various console screen functions under program control. For example, (ASCII 27) produces the commonly used ASCII **escape character**.

If *symbol* is a symbol, (CHAR-CODE *symbol*) returns the ASCII code for the first character of *symbol*'s print name. Otherwise CHAR-CODE returns NIL.

If *n* is an integer between 0 and 255 inclusive, (CODE-CHAR *n*) returns the symbol whose print name is a single character which has an ASCII code equal to *n*. Otherwise CODE-CHAR returns NIL.

CHAR-CODE and CODE-CHAR are easily defined in terms of the more general function ASCII. They are included in COMMON.LSP solely for compatibility.

```
(DEFUN ASCII (ATM)
  ((SYMBOLP ATM)
    ((PLUSP (LENGTH ATM))
      (the ASCII code for the first character of ATM's print name) ) )
  ((AND (INTEGERP ATM) (<= 0 ATM 255))
    (the character whose ASCII code is equal to ATM) ) )
(DEFUN CHAR-CODE (SYM)
  ((SYMBOLP SYM)
    (ASCII SYM) ) )

(DEFUN CODE-CHAR (N)
  ((AND (INTEGERP N) (<= 0 N 255))
    (ASCII N) ) )

$ (ASCII "A")                                          ;ASCII code for "A"
65

$ (ASCII " ")                                         ;ASCII code for blank
32

$ (ASCII 65)                                  ;Character whose ASCII code is 65
A

$ (ASCII "4")                                          ;ASCII code for "4"
52
```

CHAR-UPCASE [*symbol*] Function
CHAR-DOWNCASE [*symbol*] Function

(CHAR-UPCASE *symbol*) returns the symbol whose print name is the first character of *symbol*'s print name in uppercase.

(CHAR-DOWNCASE *symbol*) returns the symbol whose print name is the first character of *symbol*'s print name in lowercase.

If their first argument is not a symbol, these functions return NIL.

```
(DEFUN CHAR-UPCASE (SYM)
  ((SYMBOLP SYM)
    ((LOWER-CASE-P SYM)
      (ASCII (- (ASCII SYM) 32)) )
    (CHAR SYM 0) ) )

(DEFUN CHAR-DOWNCASE (SYM)
  ((SYMBOLP SYM)
    ((UPPER-CASE-P SYM)
      (ASCII (+ (ASCII SYM) 32)) )
    (CHAR SYM 0) ) )

$ (CHAR-UPCASE "ABC")
A

$ (CHAR-UPCASE "abc")
A

$ (CHAR-DOWNCASE "ABC")
\a

$ (CHAR-DOWNCASE "abc")
\a
```

5.5.2 String Functions

The functions in this subsection treat the print name of a symbol or a number as a string. Functions are provided for concatenating, comparing, trimming, and searching strings, for extracting substrings, and for converting the case of strings.

The print name of symbols are stored explicitly as character strings. The print name of numbers must be derived implicitly from their value, which is stored in binary. When called with numerical arguments, the string functions automatically create a character string equivalent to the numbers' value based on the current output radix base (see *PRINT-BASE* in Section 5.8.5). Thus, the print name of a number, as seen by the string functions, changes when the output radix base changes.

UNPACK [*atom*] Function

(UNPACK *atom*) returns a list of symbols whose one character print names are the characters in the printed representation of *atom*. If *atom* is nonatomic, UNPACK returns NIL. UNPACK is called EXPLODE in some dialects of LISP.

```
(DEFUN UNPACK (ATM)
  ((SYMBOLP ATM)
    (a list of symbols whose print names consist of the characters
        in ATM's print name) )
  ((NUMBERP ATM)
    (a list of symbols whose print names consist of the digits in
        the printed represented of ATM) ) )

$ (UNPACK 'ABCDE)                                    ;List of char symbols
(A B C D E)

$ (SETQ FOO -216)                                        ;Set FOO to -216
-216

$ (SETQ *PRINT-BASE* 16)                            ;Set output radix to 16
10

$ (UNPACK FOO)                                    ;List of hexadecimal numerals
(- \0 D \8)
```

PACK [*list*] Function
PACK* [$atom_1$, $atom_2$, ..., $atom_n$] Function

(PACK *list*) returns the symbol whose print name consists of the concatenated print names of the atoms in *list*. PACK is called COMPRESS or IMPLODE in some dialects of LISP.

(PACK* $atom_1$ $atom_2$ ... $atom_n$) returns the symbol whose print name consists of the concatenated print names of $atom_1$ through $atom_n$. PACK* is merely a no-spread version of PACK (i.e. it takes an arbitrary number of atoms rather than a list of atoms).

The current output radix base is used to determine the print name of numbers. PACK and PACK* always return a symbol, even if the resulting print name contains only digits.

```
(DEFUN PACK (LST)
  ((ATOM LST)  "")
  ((SYMBOLP (CAR LST))
    (the symbol whose print name consists of the print name of (CAR LST)
        concatenated with (PACK (CDR LST))) )
  ((NUMBERP (CAR LST))
    (the symbol whose print name consists of the digits in the printed
        representation of (CAR LST) concatenated with (PACK (CDR LST))) )
  (PACK (CDR LST)) )

(DEFUN PACK* LST
  (PACK LST) )

$ (PACK '(WARE HOUSE))                                    ;Pack list of symbols
WAREHOUSE

$ (PACK* 'WARE 'HOUSE)                                    ;Pack symbols
WAREHOUSE

$ (PACK* 123456789)                                  ;Convert numbers to strings
|123456789|
```

SUBSTRING [*atom*, *n*, *m*] Function
CHAR [*atom*, *n*] Function

If *atom* is either a symbol or a number, *n* and *m* are nonnegative integers, and *n* *m*, (SUBSTRING *atom n m*) returns the symbol whose print name consists of the *n*th through *m*th characters of *atom*'s print name, where the first character of the print name is the *zeroth* character.

All values of *n* less than 0 are equivalent to *n* being equal to 0. If *m* is omitted, is not a nonnegative integer, or is greater than the number of characters in *atom*'s print name, it is equivalent to *m* being equal to the number of characters in the print name. If *n* is greater than or equal to the number of characters in the print name or if $n > m$, SUBSTRING returns "" (i.e. the symbol whose print name is the empty or null string).

If *atom* is either a symbol or a number and *n* is a nonnegative integer, (CHAR *atom n*) returns the symbol whose print name is the *n*th character of *atom*'s print name, where the first character of the print name is the *zeroth* character. If *n* is neither zero nor a positive integer or if *atom*'s print name has *n* or fewer characters, CHAR returns NIL.

These functions always return a symbol even though their first argument is a number.

```
(DEFUN SUBSTRING (ATM N M)
  ((AND (ATOM ATM) (INTEGERP N))
    ((MINUSP N)
      (SUBSTRING ATM 0 M) )
    (PACK (SUBLIST (UNPACK ATM) N M)) ) )

(DEFUN CHAR (ATM N)
  (SUBSTRING ATM N N) )

$ (SUBSTRING 'ABCDEFG 2 4)                          ;2nd thru 4th letters
CDE

$ (SUBSTRING 'ABCDEFG 2)                            ;2nd thru last letters
CDEFG

$ (SUBSTRING 1000 0)                              ;Convert number to symbol
|1000|

$ (CHAR 'ABCDEFG 3)                                        ;Third element
D

$ (CHAR 5432 0)                                           ;Zeroth element
\5                                              ;CHAR always returns a symbol

$ (CHAR 'ABCDEFG 10)                                        ;Beyond end
NIL
```

STRING= [$atom_1$, $atom_2$, *flag*] Function
STRING-EQUAL [$atom_1$, $atom_2$] Library macro

If $atom_1$'s print name is lexicographically equal to $atom_2$'s print name, (STRING= $atom_1$ $atom_2$) returns T, otherwise it returns NIL. If *flag* is nonNIL, (STRING= $atom_1$ $atom_2$ *flag*) makes the same comparison but with no distinction between upper and lower case letters.

STRING-EQUAL makes the same comparison as STRING= but with no distinction between upper and lower case letters. Thus, STRING-EQUAL provides no additional functionally and is included in COMMON.LSP solely for compatibility.

```
(DEFUN STRING= (ATM1 ATM2 FLAG)
  ((AND (ATOM ATM1) (ATOM ATM2))
    ((NULL FLAG)
      (return T if ATM1's print name is lexicographically equal to
        ATM2's print name) )
    (return T if ATM1's print name is lexicographically, by
       ignoring case, equal to ATM2's print name) ) )

(DEFMACRO STRING-EQUAL (ATM1 ATM2)
  (LIST 'STRING= ATM1 ATM2 T) )

$ (STRING= "FAST" "FASTER")                          ;Unequal string lengths
NIL

$ (STRING= 100 |100|)                                ;Have same print names
T

$ (STRING= "Fast" "FAST")                                ;Case sensitive
NIL

$ (STRING= "Fast" "FAST" T)                              ;Case insensitive
T
```

STRING< [$atom_1$, $atom_2$, *flag*] Function
STRING> [$atom_1$, $atom_2$, *flag*] Function
STRING<= [$atom_1$, $atom_2$, *flag*] Function
STRING>= [$atom_1$, $atom_2$, *flag*] Function
STRING/= [$atom_1$, $atom_2$, *flag*] Function

If $atom_1$'s print name is lexicographically less than $atom_2$'s print name, (STRING< $atom_1$ $atom_2$) returns the position of the first character at which the print names fail to match, otherwise it returns NIL. If *flag* is nonNIL, (STRING< $atom_1$ $atom_2$ *flag*) makes the same comparison with no distinction between upper and lower case letters.

The other four functions also perform a **lexicographical comparison** and return NIL or the position of the first mismatch. STRING> performs a greater than test, STRING<= performs a less than or equal test, STRING>= performs a greater than or equal test, and STRING/= performs a not equal test.

```
(DEFUN STRING< (ATM1 ATM2 FLAG)
  ((AND (ATOM ATM1) (ATOM ATM2))
    ((NULL FLAG)
      ((if ATM1's print name is lexicographically less than ATM2's print name)
        (the position of the first mismatch)) )
    ((if ATM1's print name is lexicographically, by ignoring case,
         less than ATM2's)
      (the position of the first mismatch)) ) )

$ (STRING< 'DOG 'CAT)                                    ;DOG ordered after CAT
NIL

$ (STRING< 'CAT 'DOG)                                    ;CAT ordered before DOG
0

$ (STRING< 'DOG 'DOGGY)                                  ;Differ at 3rd letter
3

$ (STRING<= 'DOG 'DOG)                                   ;Have same print name
3
```

STRING-LESSP [$atom_1$, $atom_2$] Library macro
STRING-GREATERP [$atom_1$, $atom_2$] Library macro
STRING-NOT-GREATERP [$atom_1$, $atom_2$] Library macro
STRING-NOT-LESSP [$atom_1$, $atom_2$] Library macro
STRING-NOT-EQUAL [$atom_1$, $atom_2$] Library macro

These macros perform a **lexicographical comparison** of their two atomic arguments making *no* distinction between upper and lower case letters. If the comparison is true, they return the position of the first character at which the print names fail to match, otherwise they return NIL.

STRING-LESSP performs a less than test, STRING-GREATERP performs a greater than test, STRING-NOT-GREATERP performs a less than or equal test, STRING-NOT-LESSP performs a greater than or equal test, and STRING-NOT-EQUAL performs a not equal test.

These macros provide no additional functionally and are included in COMMON.LSP solely for compatibility.

```
(DEFMACRO STRING-LESSP (ATM1 ATM2)
  (LIST 'STRING< ATM1 ATM2 T) )

(DEFMACRO STRING-NOT-GREATERP (ATM1 ATM2)
  (LIST 'STRING<= ATM1 ATM2 T) )

$ (STRING-LESSP "dog" "CAT")                          ;DOG ordered after CAT
NIL

$ (STRING-LESSP "cat" "DOG")                          ;CAT ordered before DOG
0

$ (STRING-LESSP "DOG" "doggy")                        ;Differ at 3rd letter
3

$ (STRING-NOT-GREATERP "DOG" "DOG")                   ;Same print name
3
```

STRING-TRIM [*chars*, *atom*] Function
STRING-LEFT-TRIM [*chars*, *atom*] Function
STRING-RIGHT-TRIM [*chars*, *atom*] Function

If *chars* is a list of characters, (STRING-TRIM *chars atom*) returns a symbol whose print name is *atom*'s print name with the characters in *chars* stripped off both ends. If *chars* is a symbol, STRING-TRIM uses all the characters in its print name instead.

STRING-LEFT-TRIM and STRING-RIGHT-TRIM are similar to STRING-TRIM but they only strip characters off the left and right ends of *atom* respectively.

```
(DEFUN STRING-TRIM (LST ATM)
  (STRING-RIGHT-TRIM LST (STRING-LEFT-TRIM LST ATM)) )

(DEFUN STRING-LEFT-TRIM (LST ATM)
  ((EQ ATM "") ATM)
  ((MEMBER (CHAR ATM 0) LST)
    (STRING-LEFT-TRIM LST (SUBSTRING ATM 1)) )
  ATM )

(DEFUN STRING-RIGHT-TRIM (LST ATM)
  ((EQ ATM "") ATM)
  ((MEMBER (CHAR ATM (SUB1 (LENGTH ATM))) LST)
    ((= (LENGTH ATM) 1) "")
    (STRING-RIGHT-TRIM LST (SUBSTRING ATM 0 (- (LENGTH ATM) 2))) )
  ATM )

$ (STRING-TRIM '| ,.| '|,Joe Doe. |)
|Joe Doe|

$ (STRING-TRIM '(| | |,| |.|) '|,Joe Doe. |)
|Joe Doe|

$ (STRING-LEFT-TRIM '| ,.| '|,Joe Doe. |)
|Joe Doe. |

$ (STRING-RIGHT-TRIM '| ,.| '|,Joe Doe. |)
|,Joe Doe|
```

STRING-UPCASE [*atom*] Function
STRING-DOWNCASE [*atom*] Function
STRING-CAPITALIZE [*atom*] Function

(STRING-UPCASE *atom*) returns the symbol whose print name is the same as *atom*'s print name except that all lowercase characters are converted to uppercase.

(STRING-DOWNCASE *atom*) returns the symbol whose print name is the same as *atom*'s print name except that all uppercase characters are converted to lowercase.

(STRING-CAPITALIZE *atom*) returns the symbol whose print name is the same as *atom*'s print name except that the first, and only the first, letter of each word is capitalized.

If their argument is not an atom, these functions return NIL.

```
(DEFUN STRING-UPCASE (ATM)
  ((ATOM ATM)
    (PACK (MAPCAR 'CHAR-UPCASE (UNPACK ATM))) ) )

(DEFUN STRING-DOWNCASE (ATM)
  ((ATOM ATM)
    (PACK (MAPCAR 'CHAR-DOWNCASE (UNPACK ATM))) ) )

$ (STRING-UPCASE "LISP is recursive!")
|LISP IS RECURSIVE!|                                    ;All upper case

$ (STRING-DOWNCASE "LISP is recursive!")
|lisp is recursive!|                                    ;All lower case

$ (STRING-CAPITALIZE "LISP is recursive!")
|Lisp Is Recursive!|                          ;All first letters capitalized
```

FINDSTRING [$atom_1$, $atom_2$, n] Function

(FINDSTRING $atom_1$ $atom_2$ n) returns the position of the first occurrence of $atom_1$'s print name in $atom_2$'s print name, where zero corresponds to the first character of a print name. If n is zero or a positive integer, the search begins with the nth character of $atom_2$. If $atom_1$'s print name is not found, FINDSTRING returns NIL.

If its first or second argument is not an atom, FINDSTRING returns NIL.

```
(DEFUN FINDSTRING (ATM1 ATM2 N)
  ((ATOM ATM1)
    ((ATOM ATM2)
     ((PLUSP N)
       ((FINDSTRING ATM1 (SUBSTRING ATM2 N))
         (+ N (FINDSTRING ATM1 (SUBSTRING ATM2 N))) ) )
     ((if ATM1 occurs as a substring in ATM2)
       (the position of ATM1's first occurrence in ATM2) ) ) ) )

$ (FINDSTRING 'XYZ 'ABCXYZDEFXYZGHI)                ;Index of first occurrence of XYZ
3

$ (FINDSTRING 'XYZ 'ABCXYZDEFXYZGHI 4)              ;First occurrence after 4th letter
9

$ (FINDSTRING 'XYZ 'ABCDEFGHI)                                    ;XYZ not found
NIL
```

PRINT-LENGTH [*atom*] Function

If *atom* is an atom, (PRINT-LENGTH *atom*) returns the number of characters required to print *atom*, taking into account the current value of the control variables *PRINT-ESCAPE* and *PRINT-BASE* (see Section 5.8.5).

If its argument is not an atom, PRINT-LENGTH returns NIL.

```
(DEFUN PRINT-LENGTH (ATM)
  ((ATOM ATM)
    (LENGTH (UNPACK ATM)) ) )

$ (PRINT-LENGTH 'MULISP)                                    ;Print length of symbol
6

$ (PRINT-LENGTH -100)                                       ;Print length of number
4                                                                    ;in decimal

$ (SETQ *PRINT-BASE* 16)                                      ;Print hexadecimal
10

$ (PRINT-LENGTH -100)                                       ;Print length of number
3                                          ;in hexadecimal (-100 is -64 in hex)

$ (SETQ *PRINT-BASE* 10)                                        ;Print decimal
10

$ (PRINT-LENGTH '|Live Aid|)                                ;Print length of symbol
10

$ (SETQ *PRINT-ESCAPE* NIL)                                 ;Print w/o escape chars
NIL

$ (PRINT-LENGTH '|Live Aid|)                                      ;New print length
8

$ (SETQ *PRINT-ESCAPE* T)                                  ;Print with escape chars
T
```

5.5.3 Symbol Creation

The function and control variables in this subsection are for creating symbols with new print name strings. They are defined in the file COMMON.LSP.

GENSYM [*atom*] — Library function
GENSYM-PREFIX: "G" — Library control variable
GENSYM-COUNT: 0 — Library control variable

(GENSYM) creates a symbol by concatenating the values of *GENSYM-PREFIX* and *GENSYM-COUNT*, increments *GENSYM-COUNT*, and returns the newly created symbol.

If *atom* is a symbol, (GENSYM *atom*) sets *GENSYM-PREFIX* to *atom* before proceeding to create a symbol as described above.

If *atom* is zero or a positive integer, (GENSYM *atom*) sets *GENSYM-COUNT* to *atom* before proceeding to create a symbol as described above.

Note that there is no guarantee that the symbol created by GENSYM is actually a unique new symbol distinct from the other symbols in the system.

```
$ (GENSYM)                                          ;Create a new symbol
G0

$ (GENSYM "GENSYM-")                                ;Change the GENSYM prefix
GENSYM-1

$ (GENSYM 7)                                        ;Change the GENSYM counter
GENSYM-7

$ (GENSYM)                                          ;Create a new symbol
GENSYM-8
```

5.6 Control Constructs

The special forms and macros in this section provide *explicit* control over the evaluation of forms in function definitions. The tree structure of definitions can also be used to provide *implicit* control over the evaluation of forms.

Like most LISP systems, **muLISP** uses an **implicit** PROGN to sequentially evaluate the forms making up a function definition. **muLISP** also uses an **implicit** COND to provide for **conditional branching** in definitions. This implicit control mechanism make definitions more readable, concise, and efficient. See PROGN for a detailed discussion of the implicit control mechanism.

5.6.1 Sequencing

The control constructs in this subsection are for sequentially evaluating a list of forms. They differ primarily in the value returned.

Because of the implicit control mechanism used to evaluate function definitions, explicit use of PROGN is rarely necessary. However, the way forms are evaluated by PROGN is identical to the way forms are evaluated by the **muLISP** interpreter when processing a function definition. Thus the following information is essential to understanding how functions are applied and evaluated.

PROGN [$form_1$, $form_2$, ..., $form_n$] Special form

(PROGN $form_1$ $form_2$ ... $form_n$) sequentially evaluates the forms beginning with $form_1$ and returns the value of the *last* form evaluated. Evaluation of $form_i$ depends on its list structure as follows:

If $form_i$ is an atom or if the car of $form_i$ is an atom or a LAMBDA body, PROGN evaluates $form_i$ using EVAL (see Section 5.7.1). If the car of $form_i$ has not been defined, an Undefined Function error break occurs (see Section 4.5). Obviously atomic PROGN forms are useful only if they are the *last* form evaluated.

If the caar of $form_i$ is an atom or a LAMBDA body, PROGN evaluates the car of $form_i$. If the result is NIL, PROGN proceeds to evaluate the next form. If the result is nonNIL and the cdr of $form_i$ is empty, PROGN abandons the remaining forms and returns the nonNIL result. If the result is nonNIL and the cdr of $form_i$ is *non*empty, PROGN abandons the remaining forms and evaluates the cdr of $form_i$ using an implicit PROGN. This conditional branch mechanism, called an **implicit** COND, causes a permanent fork in a program's flow of control.

If the caar of $form_i$ is not an atom, PROGN recursively uses an implicit PROGN to evaluate $form_i$ itself as a list of forms. PROGN then proceeds to evaluate the next form. Note that the first subform of $form_i$ will evaluate as an implicit COND. This **recursive implicit** COND does *not* cause a permanent fork in a program's flow of control.

```
(DEFMACRO PROGN LST
  (LIST 'eval-progn (LIST 'QUOTE LST) NIL) )

(DEFUN eval-progn (LST RSLT)
  ((NULL LST)  RSLT)
  ((OR (ATOM (CAR LST)) (ATOM (CAAR LST)) (EQ (CAAAR LST) 'LAMBDA))
    (eval-progn (CDR LST) (EVAL (CAR LST))) )
  ((OR (ATOM (CAAAR LST)) (EQ (CAAAAR LST) 'LAMBDA))
    ((SETQ RSLT (EVAL (CAAR LST)))
      (eval-progn (CDAR LST) RSLT) )
    (eval-progn (CDR LST)) )
  (eval-progn (CDR LST) (eval-progn (CAR LST))) )
```

PROG1 [$form_1$, $form_2$, ..., $form_n$] Special form
PROG2 [$form_1$, $form_2$, ..., $form_n$] Library macro

(PROG1 $form_1$ $form_2$ ... $form_n$) evaluates $form_1$, evaluates the remaining forms using an implicit PROGN, and returns the result of the evaluation of $form_1$.

(PROG2 $form_1$ $form_2$ ... $form_n$) evaluates $form_1$, evaluates $form_2$, evaluates the remaining forms using an implicit PROGN, and returns the result of the evaluation of $form_2$.

Sometimes PROG1 and PROG2 can avoid having to introduce a temporary variable to store a result while other expressions are being evaluated.

PROG2 is defined in COMMON.LSP.

```
(DEFMACRO PROG1 LST
  ((NULL LST)  NIL)
  (LIST 'eval-prog1 (LIST 'QUOTE (CDR LST)) (CAR LST)) )

(DEFUN eval-prog1 (LST RSLT)
  (eval-progn LST)
  RSLT )

(DEFMACRO PROG2 (FORM1 . FORMS)
  (LIST 'PROGN FORM1 (CONS 'PROG1 FORMS)) )

$ (SETQ FOO '(A B C D))                                    ;Initialize FOO
(A B C D)

$ (PROG1 (CAR (LAST FOO)) (SETQ FOO (NBUTLAST FOO)))    ;Remove and return last element
D

$ FOO                                                      ;Last element removed
(A B C)
```

LET [*letlist*, $form_1$, $form_2$, ..., $form_n$] Library macro
LET* [*letlist*, $form_1$, $form_2$, ..., $form_n$] Library macro

If *letlist* is a list of *m* elements of the form (var_i $value_i$), (LET *letlist* $form_1$ $form_2$ ... $form_n$) evaluates the expressions $value_1$ through $value_m$ in order, and temporarily binds the resulting values to the variables var_1 through var_m. Next LET evaluates the forms $form_1$ through $form_n$ using an implicit PROGN, restores the original values to the variables var_1 through var_m, and returns the value of the last form evaluated.

If *letlist* is a list of *m* elements of the form (var_i $value_i$), (LET* *letlist* $form_1$ $form_2$ ... $form_n$) evaluates the expression $value_1$ and temporarily binds the resulting value to the variable var_1, evaluates the expression $value_2$ and temporarily binds the resulting value to the variable var_2, and so on through var_m. Next LET* evaluates the forms $form_1$ through $form_n$ using an implicit PROGN, restores the original values to the variables var_1 through var_m, and returns the value of the last form evaluated.

In short, LET and LET* operate identically, except that LET binds the variables in parallel, whereas LET* binds the variables sequentially.

LET and LET* are defined in COMMON.LSP.

```
$ (SETQ X 3)
3

$ (LET ((X 5)
        (Y X))
    (+ X Y) )
8

$ (LET* ((X 5)
         (Y X))
    (+ X Y) )
10

$ X
3
```

5.6.2 Conditionals

The control constructs in this subsection provide for conditional branching in programs based on the value of predicates.

IF [*testform*, *thenform*, *elseform*] Special form
WHEN [*testform*, $form_1$, $form_2$, ..., $form_n$] Library macro
UNLESS [*testform*, $form_1$, $form_2$, ..., $form_n$] Library macro

If the value of *testform* is nonNIL, (IF *testform thenform*) evaluates and returns *thenform*, otherwise it returns NIL. If the value of *testform* is nonNIL, (IF *testform thenform elseform*) evaluates and returns *thenform*, otherwise it evaluates and returns *elseform*. IF is equivalent to the **if-then-else** construct found in many programming languages.

If the value of *testform* is nonNIL, (WHEN *testform* $form_1$ $form_2$ ... $form_n$) evaluates the remaining forms using an implicit PROGN and returns the value of the last form evaluated. If *testform* is NIL, WHEN returns NIL without evaluating the remaining forms.

If the value of *testform* is NIL, (UNLESS *testform* $form_1$ $form_2$ ... $form_n$) evaluates the remaining forms using an implicit PROGN and returns the value of the last form evaluated. If *testform* is nonNIL, UNLESS returns NIL without evaluating the remaining forms.

WHEN and UNLESS are defined in COMMON.LSP. Several formal definitions in this chapter provide good examples using the IF control construct.

```
(DEFMACRO IF (TEST THEN ELSE)
  (LIST 'PROGN (LIST TEST THEN) ELSE) )

(DEFMACRO WHEN (TEST . BODY)
  (LIST 'IF TEST (CONS 'PROGN BODY)) )

(DEFMACRO UNLESS (TEST . BODY)
  (LIST 'IF TEST NIL (CONS 'PROGN BODY)) )
```

COND [$cond_1$, $cond_2$, ..., $cond_n$] Special form

(COND $cond_1$ $cond_2$ ... $cond_n$) successively evaluates the car of each cond form until a nonNIL value is encountered or all the forms are evaluated. If the result of the car of $cond_i$ is nonNIL and the cdr of $cond_i$ is empty, COND abandons the remaining cond forms and returns the nonNIL result. If the result of the car of $cond_i$ is nonNIL and the cdr of $cond_i$ is *non*empty, COND abandons the remaining cond forms and evaluates the cdr of $cond_i$ using an implicit PROGN. If the car of each cond form is evaluated and all return NIL, COND returns NIL.

The implicit COND has greatly diminished the need to use COND explicitly. However, as the example shows, COND is useful when making a conditional statement within nested function calls.

```
(DEFMACRO COND LST
  (LIST 'eval-cond (LIST 'QUOTE LST) NIL) )

(DEFUN eval-cond (LST RSLT)
  ((NULL LST)  NIL)
  ((SETQ RSLT (EVAL (CAAR LST)))
    (eval-progn (CDAR LST) RSLT) )
  (eval-cond (CDR LST)) )

$ (SETQ NUM -3)
-3

$ (SETQ SIGN (COND
      ((PLUSP NUM)  'POSITIVE)
      ((MINUSP NUM)  'NEGATIVE)
      ((ZEROP NUM)  'ZERO)
      ('NONNUMBER) ) )
NEGATIVE
```

CASE [*keyform*, $case_1$, ..., $case_n$] Library macro
OTHERWISE Library keyword

(CASE keyform $case_1$ ... $case_n$) evaluates *keyform* to produce a key. This key is compared with the car of each case clause looking for a match. A match occurs if the car of the case clause is the symbol T or the keyword OTHERWISE, if it is a nonNIL atom and key is EQL to the atom, or if it is a list (i.e. NIL or a cons) and key is EQL to any member of the list. If a match is found, CASE evaluates the cdr of the matching case clause as a list of forms using an implicit PROGN. If the cdr of the matching case clause is empty, CASE returns NIL. If no match is found, CASE returns NIL.

CASE is defined in COMMON.LSP.

```
(DEFMACRO CASE (KEYFORM . CASES)
  (LIST
    (LIST 'LAMBDA '(KEY) (CONS 'COND
      (MAPCAR '(LAMBDA (CASE) (CONS
          (IF (MEMBER (CAR CASE) '(T OTHERWISE))
              T
              (LIST (IF (ATOM (CAR CASE)) 'EQL 'MEMBER)
                    KEY
                    (LIST 'QUOTE (CAR CASE))) )
          (OR (CDR CASE) '(NIL))) )
        CASES)))
    KEYFORM) )

$ (CASE (+ 2 3)
     (0 'ZERO)
     (1 'ONE)
     ((2 4 6 8) 'EVEN)
     ((3 5 7 9) 'ODD)
     (T '?) )
ODD
```

5.6.3 Iteration

Implicit PROGN and implicit COND are powerful function body evaluation mechanisms but they do not provide for *nonrecursive* program control structures. Such iterative capability is provided by the control constructs in this subsection.

LOOP [$form_1$, $form_2$, ..., $form_n$] Special form

(LOOP $form_1$ $form_2$ ... $form_n$) repeatedly evaluates the forms in sequential order until a nonNIL implicit COND is encountered. LOOP then evaluates the body of the implicit COND and returns the result.

LOOP evaluates its forms sequentially in a manner identical to the evaluation of forms by PROGN. However, if all of a LOOP's forms are evaluated without a conditional having been satisfied, evaluation begins again with the first form.

Any number of implicit CONDs can occur within a loop, and they can be located at the beginning, middle, or end of a loop. This flexible loop control construct preserves the structured nature and clarity of programs while improving their performance.

The source files distributed with **muLISP** contain numerous examples using the LOOP control construct. Although this formal definition is recursive, LOOP is actually a nonrecursive machine language routine.

```
(DEFMACRO LOOP LST
  (LIST 'eval-loop (LIST 'QUOTE LST) NIL) )

(DEFUN eval-loop (LST RSLT)
  ((NULL LST)
    (eval-loop FORM-LST) )
  ((OR (ATOM (CAR LST)) (ATOM (CAAR LST)) (EQ (CAAAR LST) 'LAMBDA))
    (eval-loop (CDR LST) (EVAL (CAR LST))) )
  ((OR (ATOM (CAAAR LST)) (EQ (CAAAAR LST) 'LAMBDA))
    ((SETQ RSLT (EVAL (CAAR LST)))
      (eval-progn (CDAR LST) RSLT) )
    (eval-loop (CDR LST)) )
  (eval-loop (CDR LST) (eval-progn (CAR LST))) )

$ (SETQ NUM 1)
1

$ (LOOP ((> NUM 4)) (PRINT (* NUM NUM)) (INCQ NUM))                    ;Display squares
1
4
9
16
```

DO [*letlist*, $form_1$, $form_2$, ..., $form_n$] Library macro
DO* [*letlist*, $form_1$, $form_2$, ..., $form_n$] Library macro

If *letlist* is a list of *m* elements of the form (var_i $value_i$ $step_i$), (DO *letlist* $form_1$ $form_2$ ... $form_n$) evaluates the expressions $value_1$ through $value_m$ in order, and temporarily binds the resulting values to the variables var_1 through var_m. Next DO evaluates the forms $form_1$ through $form_n$ using an implicit PROGN. If all the forms are evaluated without encountering a nonNIL implicit COND, DO evaluates the expressions $step_1$ through $step_m$ in order, and temporarily binds the resulting values to the variables var_1 through var_m. DO continues to evaluate the forms and rebind the variables until a nonNIL implicit COND is encountered. Then DO evaluates the body of the implicit COND, restores the original values to the variables var_1 through var_m, and returns the result of the implicit COND.

DO* operates identically to DO except that DO* binds the variables sequentially rather than in parallel.

In **muLISP** any or all of the forms in a DO or DO* loop can have conditional exit tests, whereas in Common LISP $form_1$ is the one and only conditional exit test.

DO and DO* are defined in COMMON.LSP.

```
$ (DO ((LST '(A B C D) (CDR LST))
       (RSLT NIL (CONS (CAR LST) RSLT)))
    ((NULL LST) RSLT) )
(D C B A)                                    ;Reverse of list

$ (DO ((X 1 (ADD1 X))
       (Y 0 (+ Y X)))
    ((= X 10) Y) )
45                                           ;Sum of 1 thru 9
```

DOLIST [(*var listform resultform*), $form_1$, ..., $form_n$] Library macro
DOTIMES [(*var countform resultform*), $form_1$, ..., $form_n$] Library macro

(DOLIST (*var listform resultform*) $form_1$ ... $form_n$) evaluates *listform*, binds *var* to the first element of *listform*, and evaluates the forms $form_1$ through $form_n$ using an implicit PROGN. Next DOLIST binds *var* to the second element of *listform*, reevaluates the forms, and so on until the list is exhausted. Finally, DOLIST binds *var* to NIL, evaluates *resultform* if given, restores *var* to its original value, and returns the value of *resultform* if given or NIL otherwise.

(DOTIMES (*var countform resultform*) $form_1$ ... $form_n$) evaluates *countform* and binds *var* to 0. If *var* < *countform*, DOTIMES evaluates the forms $form_1$ through $form_n$ using an implicit PROGN. Next DOTIMES binds *var* to 1 and if *var* < *countform*, reevaluates the forms, and so on until *var* *countform*. Finally, DOTIMES evaluates *resultform* if given, restores *var* to its original value, and returns the value of *resultform* if given or NIL otherwise.

DOLIST and DOTIMES are defined in COMMON.LSP.

```
(DEFMACRO DOLIST ((VAR LISTFORM RSLTFORM) . BODY)
  (LET ((DOLIST (GENSYM)))
       (LIST 'LET
             (LIST (LIST DOLIST LISTFORM))
             (LIST* 'LOOP
                    (LIST (LIST 'ATOM DOLIST) RSLTFORM)
                    (LIST 'SETQ VAR (LIST 'POP DOLIST))
                    BODY))) )

$ (DOLIST (X '(A P P L E)) (PRIN1 X))                    ;Display letters in list
APPLE

$ (DOTIMES (X 5)                                        ;Display square of 1, 2, 3, & 4
          (PRIN1 X) (SPACES 2) (PRINT (* X X)) )
0  0
1  1
2  4
3  9
4  16
NIL
```

5.6.4 Non-local Exits

The special forms in this subsection provide a facility for non-local exits from nested function calls. Such exits are especially useful for handling error conditions.

CATCH [*tag*, $form_1$, ..., $form_n$] Special form

After evaluating *tag*, (CATCH *tag* $form_1$ ... $form_n$) evaluates the forms beginning with $form_1$ using an implicit PROGN and returns the result of the last form evaluated, unless an uncaught throw occurs during the evaluation of the forms. If such a throw occurs, and if *tag* is NIL or EQ to *THROW-TAG* (i.e. the tag specified by the throw), CATCH deactivates the throw and immediately returns the value specified by the throw. If a throw occurs that does not match *tag*, the throw continues uncaught by this catch.

If the catch tag is NIL, all throws are caught. The throw responsible can be determined by examining the value of the variable *THROW-TAG* (see THROW).

The default read-eval-print loop catches throws to the tags RETURN and DRIVER. The top-level loop catches all throws not otherwise caught (see Section 5.10.1). See the definition of the function DRIVER for a good example of the use of CATCH.

```
(DEFMACRO CATCH LST
  ((NULL LST)  NIL)
  (LIST 'eval-catch (CAR LST) (LIST 'PROGN (CDR LST))) )

(DEFUN eval-catch (TAG RSLT)
  ((if no throw is active)  RSLT)
  ((OR (NULL TAG) (EQ TAG *THROW-TAG*))
    (deactivate throw)
    (return throw value) )
  (continue throw) )
```

UNWIND-PROTECT [$form_1$, $form_2$, ..., $form_n$] Special form

(UNWIND-PROTECT $form_1$ $form_2$ ... $form_n$) evaluates $form_1$ and then evaluates the rest of the forms using an implicit PROGN even if evaluation of $form_1$ was terminated by a throw. When the implicit PROGN is complete and if the evaluation of $form_1$ terminated normally, UNWIND-PROTECT returns the result of evaluating $form_1$. If the evaluation of $form_1$ was terminated by a throw, UNWIND-PROTECT continues the throw.

UNWIND-PROTECT is useful for those situations where it is essential to properly terminate or "clean-up" after evaluating $form_1$ no matter what caused it to terminate. For example, the following code could be used to ensure that the precision is restored to its original value following the computation of π to a high accuracy even if the computation was aborted by a keyboard interrupt or a Memory Full error break.

The formal definition of the function DRIVER provides another example of the use of UNWIND-PROTECT (see Section 5.10.2).

```
$ (DEFUN COMPUTE-PI (NEW-PRECISION
      OLD-PRECISION)
    (SETQ OLD-PRECISION (PRECISION NEW-PRECISION))
    (UNWIND-PROTECT (PI)
                    (PRECISION OLD-PRECISION)) )
```

THROW [*tag, form*] Special form
RETURN [*form*] Special form
THROW-TAG Variable

(THROW *tag form*) evaluates *tag* and *form*, sets the variable *THROW-TAG* to the value of *tag*, terminates function body evaluation, and unwinds the variable and control stacks back to the most recent CATCH having a catch label EQ to the value of *tag* or having a catch label of NIL. CATCH then returns *form* as its value. If not caught by the program or the read-eval-print loop, control returns to the top-level loop (see Section 5.10.1).

(RETURN *form*) evaluates *form*, terminates function body evaluation, and unwinds the variable and control stacks back to the call on the function or LAMBDA body whose definition contains the call on RETURN. This call returns the value of *form*. Since there is an implicit catch of RETURN around all function calls, RETURN is implemented by making a throw tagged RETURN.

Although THROW and RETURN are special forms, their arguments are evaluated *before* the throw is made. Besides error handling, CATCH and THROW are useful for exiting a nest of control constructs without actually leaving the function as RETURN does.

The formal definition of the function BREAK provides a less contrived example of the use of THROW (see Section 5.10.3).

```
(DEFUN THROW (TAG FORM)
  (SETQ *THROW-TAG* TAG)
  (unwind the variable and control stacks until caught by a CATCH)
  (the CATCH then returns FORM) )

(DEFUN RETURN (FORM)
  (THROW 'RETURN FORM) )
```

RESTART [ ] Function
SYSTEM [*n*] Function

(RESTART) closes all open files, abandons the current environment, and initiates a fresh **muLISP** system. All variable bindings, user-defined functions, and property values in the current environment are lost.

(SYSTEM) generates a throw tagged SYSTEM. If not caught by a program and the top-level loop catches a throw to SYSTEM, it closes all open files, terminates **muLISP**, and returns control to the host operating system.

If *n* is an integer between 0 and 255 inclusive, (SYSTEM *n*) returns an **exit code** (also called a **return code**) of *n* to its host process.

The exit code can be interrogated by the MS-DOS batch command IF ERRORLEVEL (see the Batch Processing Commands section of the MS-DOS Reference Manual for details). If *n* is omitted, **muLISP** returns the default exit code stored in the **muLISP** base page (see Appendix G).

```
(DEFUN RESTART ()
  (close all currently open files)
  (abandon current environment)
  (restore the initial start-up environment)
  (start the default read-eval-print loop) )

(DEFUN SYSTEM (BYTE)
  (close all currently open files)
  ((AND (INTEGERP BYTE) (<= 0 BYTE 255))
    (exit to host returning the exit code BYTE) )
  (exit to the host returning default exit code) )

$ (RESTART)                                    ;Restart muLISP

$ (SYSTEM)                                       ;Quit muLISP

A>                                             ;The DOS prompt
```

5.6.5 Subprocess Invocation

The functions in this subsection suspend **muLISP** and temporarily turn control over to the operating system or another program.

EXECUTE [*program*, *command*] Function
DOS [*command*] Library function

(EXECUTE *program command*) compacts and suspends **muLISP**, then loads and executes *program* passing it *command* as an argument. When *program* terminates, control reverts to **muLISP** and EXECUTE returns *program*'s exit code. If *program* is not found, EXECUTE returns NIL.

(DOS *command*) invokes the MS-DOS **command processor** (see the MS-DOS Reference Manual) to process *command* just as if *command* were entered following the MS-DOS prompt. (DOS) suspends **muLISP** and invokes the MS-DOS command processor which displays the MS-DOS prompt. MS-DOS commands can be entered until an EXIT command returns control to **muLISP**. DOS is defined in COMMON.LSP.

Using EXECUTE and DOS, the full range of facilities provided by MS-DOS and application programs can be accessed, all from within **muLISP** without abandoning the current **muLISP** environment. Communication with the "child" process can be accomplished through the use of source files, memory image files, and/or exit codes. For instance, the last example below starts running another rendition of **muLISP** and "passes" it the file EXPERT.

```
(DEFUN DOS (COMMAND)
  (EXECUTE (GETSET 'COMSPEC) (IF COMMAND (PACK* "/C " COMMAND))) )

$ (DOS "DIR B:/W")                                  ;Display drive B's directory
<display of drive B's directory>
0                                                     ;DOS returns exit code

$ (DOS)                                          ;Suspend muLISP, start-up MS-DOS
MS-DOS Version 2.10                                 ;Displays MS-DOS prompt

A>EXIT                                            ;Issue MS-DOS EXIT command
0                                                     ;DOS resumes execution

$ (EXECUTE "B:MULISP.COM" "EXPERT")
```

5.7 Interpreter

The interpreter is responsible for executing programs by evaluating forms and applying functions.

5.7.1 Form Evaluation

The primitives in this subsection are for accessing the interpreter's form evaluation mechanism.

EVAL [*form*] Function

If *form* is a symbol, (EVAL *form*) returns the contents of its value cell. Since symbols are auto-quoted when they are created (see Section 4.1.1), they evaluate to themselves until bound to another value. NIL is a constant that always evaluates to itself. Thus NIL need not be quoted.

If *form* is a number, (EVAL *form*) returns the number. Thus numbers are constants and need not be quoted.

If *form* is a cons whose car is the name of an **eval function** (see Section 5.3.3), (EVAL *form*) evaluates each element of the cdr of *form* and applies the eval function to the results. Note that the arguments of eval functions are evaluated and only the resulting values are passed to the function. Eval functions are by far the most common function type used in LISP programs.

If *form* is a cons whose car is a LAMBDA body, (EVAL *form*) evaluates each element of the cdr of *form* and applies the LAMBDA body to the results. LAMBDA bodies are lists whose car is the symbol LAMBDA, whose cadr is an argument list, and whose cddr is a list of forms. Note that the arguments of LAMBDA bodies are evaluated and only the resulting values are passed to the LAMBDA body.

If *form* is a cons whose car is the name of a **no-eval function** (see Section 5.3.3), (EVAL *form*) applies the no-eval function to the cdr of *form*. Note that the arguments of no-eval functions are *not* evaluated and the arguments themselves are passed to the function. Since no-eval functions can lead to variable name conflicts, it is usually preferable to use macros instead of no-eval functions.

If *form* is a cons whose car is the name of a **macro** (see Section 5.3.3), (EVAL *form*) applies its macro definition to *form* and then evaluates the form resulting from the **macro expansion**. Macro calls in function definitions can be expanded when the function is defined or when it is evaluated (see Section 5.7.5).

If *form* is a cons whose car is the name of a **special form** (see Section 5.3.3), (EVAL *form*) applies the machine language definition of the special form to the cdr of *form* without evaluating the elements of the latter. Users can not define special forms except in machine language (see Chapter 9).

If *form* is a cons whose car is neither a symbol nor a LAMBDA body, (EVAL *form*) generates an Undefined Function error break by applying *UNDEFINED-HOOK* to *form*.

```
(DEFUN EVAL (FORM)
  ((SYMBOLP FORM)
    ((NULL FORM)  NIL)
    (CAR FORM) )
  ((NUMBERP FORM)  FORM)
  ((SYMBOLP (CAR FORM))
    ((EVAL-FUNCTION-P (CAR FORM))
      (APPLY (CAR FORM) (MAPCAR 'EVAL (CDR FORM))) )
    ((NO-EVAL-FUNCTION-P (CAR FORM))
      (APPLY (CAR FORM) (CDR FORM)) )
    ((MACRO-P (CAR FORM))
      (EVAL (APPLY (CAR FORM) FORM)) )
    (FUNCALL *UNDEFINED-HOOK* FORM) )
  ((FUNCTIONP (CAR FORM))
    (APPLY (CAR FORM) (MAPCAR 'EVAL (CDR FORM))) )
  (FUNCALL *UNDEFINEDHOOK* FORM) )

$ (SETQ JOHN 'MAN)                                    ;Set value of JOHN to MAN
MAN

$ (SETQ MAN 'ANIMAL)                                  ;Set value of MAN to ANIMAL
ANIMAL

$ JOHN                                                ;Value of JOHN is MAN
MAN

$ (EVAL JOHN)                                         ;Value of value of JOHN is ANIMAL
ANIMAL

$ (EVAL '(CONS 'A '(B C D)))                          ;Evaluate a form
(A B C D)

$ ((LAMBDA (N) (* N N)) 5)                            ;Apply a LAMBDA body
25
```

QUOTE [*object*] Special form
IDENTITY [*object*] Function

(QUOTE *object*) returns *object* without evaluating it. *object* can be any data object (i.e. either a symbol, a number, or a cons). QUOTE is used to prevent the evaluation of an object being passed to an eval function. However, it is unnecessary to quote NIL or numbers since they are constants.

The single quote mark (') is a primitively defined read macro that can be used as a convenient abbreviation to quote an object (see Section 5.8.3).

(IDENTITY *object*) returns *object* without evaluating it. However because IDENTITY is a function, its argument is evaluated before being passed to it. In order to recognize an implicit COND, its predicate form must be a list rather than an atom (see Section 5.6.1). Hence IDENTITY is frequently used to transform an atom into a list so that it can be used as the predicate of an implicit COND.

```
(DEFUN QUOTE (NLAMBDA (OBJ)
  OBJ ))

(DEFUN IDENTITY (OBJ)
  OBJ )

$ (SETQ DOG 'ROVER)                                    ;Assign DOG a value
ROVER

$ (UNPACK DOG)                                         ;Unpack value of DOG
(R O V E R)

$ (UNPACK (QUOTE DOG))                                 ;Unpack DOG itself
(D O G)

$ (CAR (CONS 4 7))                                     ;Car of dotted pair
4

$ (CAR '(CONS 4 7))                                    ;Car of 3 element list
CONS

$ (IDENTITY DOG)                                       ;The value of DOG
ROVER

$ (IDENTITY 'DOG)                                      ;The value of 'DOG
DOG

$ (MAPCAN 'IDENTITY '(A (B) C (D E) F))                ;Filter atoms out of list
(B D E)
```

CONSTANTP [*object*] Function
FUNCTIONP [*object*] Library function

If *object* is a **constant form**, (CONSTANTP *object*) returns T, otherwise it returns NIL. Evaluation of a constant form always returns the same value. The constant forms are NIL, numbers, and lists whose car is the symbol QUOTE. Note that T is *not* a constant form.

If *object* is a symbol or a LAMBDA body, (FUNCTIONP *object*) returns T, otherwise it returns NIL. If FUNCTIONP returns T, *object* is suitable for use as the functional argument to APPLY, FUNCALL, and the mapping functions. Note that all symbols are suitable for use as functional arguments, even though they may be undefined.

FUNCTIONP is defined in COMMON.LSP.

```
(DEFUN CONSTANTP (OBJ)
  ((NULL OBJ))
  ((NUMBERP OBJ))
  ((ATOM OBJ)  NIL)
  (EQ (CAR OBJ) 'QUOTE) )

(DEFUN FUNCTIONP (OBJ)
  ((SYMBOLP OBJ))
  ((ATOM OBJ)  NIL)
  (EQ (CAR OBJ) 'LAMBDA) )

$ (CONSTANTP ())                                    ;NIL is a constant
T

$ (CONSTANTP -237.6)                               ;Numbers are constants
T

$ (CONSTANTP '(QUOTE (A B C)))                      ;Quotes are constant
T

$ (FUNCTIONP 'FOO)                          ;FOO is a suitable function name
T

$ (FUNCTIONP '(+ 2 3))                              ;Lists are not functions
NIL

$ (FUNCTIONP '(LAMBDA (X Y) (- Y X)))       ;LAMBDA bodies are suitable functions
T
```

COMMENT [*comments*] Library macro

(COMMENT *comments*) ignores its arguments and returns NIL. The words making up the **comment** must *not* contain escape or macro characters (see Section 5.8) or they will not be accepted by the **muLISP** reader.

COMMENT provides a means of including comments in a function definition. However, such comments use code space and slow execution speed slightly.

In contrast, comments delimited by comment macro characters are ignored by the **muLISP** reader when they are read in (see Section 5.8.3). Although they require disk storage space, such comments do *not* use code space or slow execution speed.

```
(DEFMACRO COMMENT LST
  NIL )

$ (DEFUN FOO (X)
    (COMMENT The definition of FOO contains a comment that refers to itself!)
    (CONS X X) )
```

5.7.2 Function Application

The primitives in this subsection are for applying functions to arguments. When a function is applied, its **formal arguments** are bound to its **actual arguments** given in the call on the function, the **function body** is evaluated using an implicit PROGN (see Section 5.6.1), and the result is returned after the formal arguments are restored to their original value.

If a function is defined using a list, possibly empty, of symbols for its formal arguments, it is called a **spread function**. When a spread function is applied, each formal argument is bound to an actual argument (i.e. the actual arguments are *spread* over the formal arguments). If there are more formal than actual arguments, the extra formal arguments are bound to NIL. If there are more actual than formal arguments, the extra actual arguments are evaluated but the results are ignored.

If a function is defined using a symbol, rather than a list, for its formal arguments, it is called a **no-spread function**. When a no-spread function is applied, the formal argument is bound to a list of the actual arguments. Thus no-spread functions can accept an arbitrary number of actual arguments. Note that the decision to make a function spread or no-spread is independent of the decision to make it eval or no-eval.

FUNCALL [*function*, arg_1, arg_2, ..., arg_n] Function

If *function* is a function name, a macro name, or a LAMBDA body, (FUNCALL *function* arg_1 arg_2 ... arg_n) pushes the current values of *function*'s formal arguments on the variable stack, binds the formal arguments to arg_1 through arg_n, evaluates the function body using an implicit PROGN, and returns the result after the formal arguments are restored to their original value.

If *function* is neither a function name, a macro name, nor a LAMBDA body, FUNCALL generates an Undefined Function error break by applying *UNDEFINED-HOOK* to (LIST *function* arg_1 arg_2 ... arg_n).

FUNCALL is just a no-spread version of APPLY.

```
(DEFUN FUNCALL LST
  (APPLY (CAR LST) (CDR LST)) )

$ (FUNCALL 'CONS 'A '(B C D))                          ;Apply CONS
(A B C D)

$ (FUNCALL '(LAMBDA (N) (* N N)) 5)                    ;Apply LAMBDA
25
```

APPLY [*function*, arg_1, arg_2, ..., arg_n, *arglist*] Function

If *function* is a function name, a macro name, or a LAMBDA body, (APPLY *function* arg_1 arg_2 ... arg_n *arglist*) pushes the current values of *function*'s formal arguments on the variable stack, binds the formal arguments to arg_1 through arg_n and the elements of *arglist*, evaluates the function body using an implicit PROGN, and returns the result after the formal arguments are restored to their original value.

If *function* is neither a function name, a macro name, nor a LAMBDA body, APPLY generates an Undefined Function error break by applying *UNDEFINED-HOOK* to (LIST* *function* arg_1 arg_2 ... arg_n *arglist*).

```
(DEFUN APPLY (FUNC ARG1 ARG2 ... ARGS)
  (SETQ ARGS (LIST* ARG1 ARG2 ... ARGS))
  ((SYMBOLP FUNC)
    ((OR (EVAL-FUNCTION-P FUNC) (NO-EVAL-FUNCTION-P FUNC))
      (eval-body ARGS (CADR (GETD FUNC)) (CDDR (GETD FUNC))) )
    ((MACRO-P FUNC)
      (eval-body (CONS FUNC (MAPCAR '(LAMBDA (ARG) (LIST 'QUOTE ARG)) ARGS))
                 (CADR (GETD FUNC))
                 (CDDR (GETD FUNC))) )
    (FUNCALL *UNDEFINED-HOOK* (CONS FUNC ARGS)) )
  ((EQ (CAR FUNC) 'LAMBDA)
    (eval-body ARGS (CADR FUNC) (CDDR FUNC)) )
  (FUNCALL *UNDEFINED-HOOK* (CONS FUNC ARGS)) )

(DEFUN eval-body (ACTUAL FORMAL BODY)
  ((LISTP FORMAL)
    (MAPC '(LAMBDA (ARG) (PUSH (EVAL ARG) variable-stack)) FORMAL)
    (MAPC 'SET FORMAL ACTUAL)
    (PROG1 (PROGN BODY)
           (MAPC '(LAMBDA (ARG) (SET ARG (POP variable-stack))) FORMAL) ) )
  (PUSH (EVAL FORMAL) variable-stack)
  (SET FORMAL ACTUAL)
  (PROG1 (PROGN BODY) (SET FORMAL (POP variable-stack))) )

$ (APPLY 'CONS '(A (B C D)))                                    ;Apply CONS
(A B C D)

$ (APPLY 'CONS 'A '((B C D)))                                   ;Apply CONS
(A B C D)

$ (APPLY 'CONS 'A '(B C D) NIL)                                 ;Apply CONS
(A B C D)

$ (APPLY '(LAMBDA (N) (* N N)) '(5))                            ;Apply LAMBDA
25
```

5.7.3 Mapping Functions

The functions in this subsection successively apply a function to the elements or sublists of one or more lists. The different mapping functions provide alternatives for returning the results of the function applications.

MAPC [*function*, $list_1$, $list_2$, ..., $list_n$] Function
MAPL [*function*, $list_1$, $list_2$, ..., $list_n$] Function

(MAPC *function* $list_1$ $list_2$ ... $list_n$) applies *function* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until any of the lists are exhausted. MAPC returns $list_1$.

(MAPL *function* $list_1$ $list_2$ ... $list_n$) applies *function* to $list_1$ through $list_n$, then to the cdr of each list, and so on until any of the lists are exhausted. MAPL returns $list_1$.

```
(DEFUN MAPC (FUNC LST1 LST2)
  ((OR (NULL LST1) (NULL LST2))  LST1)
  (FUNCALL FUNC (CAR LST1) (CAR LST2))
  (MAPC FUNC (CDR LST1) (CDR LST2))
  LST1 )

(DEFUN MAPL (FUNC LST1 LST2)
  ((OR (NULL LST1) (NULL LST2))  LST1)
  (FUNCALL FUNC LST1 LST2)
  (MAPL FUNC (CDR LST1) (CDR LST2))
  LST1 )

$ (MAPC 'PRINT '(DOG CAT COW))                          ;Print animals
DOG
CAT
COW
(DOG CAT COW)                                           ;and return list

$ (MAPL 'PRINT '(DOG CAT COW))                          ;Print animals
(DOG CAT COW)
(CAT COW)
(COW)
(DOG CAT COW)                                           ;and return list

$ (MAPL '(LAMBDA (LST) (RPLACA LST NIL)) '(A B C D E))
(NIL NIL NIL NIL NIL)                    ;Replace elements of list with NIL
```

MAPCAR [*function*, $list_1$, $list_2$, ..., $list_n$] Function
MAPLIST [*function*, $list_1$, $list_2$, ..., $list_n$] Function

(MAPCAR *function* $list_1$ $list_2$ ... $list_n$) applies *function* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until any of the lists are exhausted. MAPCAR returns a list of the results.

(MAPLIST *function* $list_1$ $list_2$ ... $list_n$) applies *function* to $list_1$ through $list_n$, then to the cdr of each list, and so on until any of the lists are exhausted. MAPLIST returns a list of the results.

```
(DEFUN MAPCAR (FUNC LST1 LST2)
  ((OR (NULL LST1) (NULL LST2))  NIL)
  (CONS (FUNCALL FUNC (CAR LST1) (CAR LST2))
        (MAPCAR FUNC (CDR LST1) (CDR LST2))) )

(DEFUN MAPLIST (FUNC LST1 LST2)
  ((OR (NULL LST1) (NULL LST2))  NIL)
  (CONS (FUNCALL FUNC LST1 LST2)
        (MAPLIST FUNC (CDR LST1) (CDR LST2))) )

$ (MAPCAR '* '(2 3 5 7) '(2 4 6 8))                ;Pairwise multiply elements of list
(4 12 30 56)

$ (MAPLIST 'REVERSE '(A B C D))                          ;Reverse list and its tails
((D C B A) (D C B) (D C) (D))
```

MAPCAN [*function*, $list_1$, $list_2$, ..., $list_n$] Function
MAPCON [*function*, $list_1$, $list_2$, ..., $list_n$] Function

(MAPCAN *function* $list_1$ $list_2$ ... $list_n$) applies *function* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until any of the lists are exhausted. MAPCAN concatenates the results using NCONC. MAPCAN is useful for filtering out undesired elements of a list.

(MAPCON *function* $list_1$ $list_2$ ... $list_n$) applies *function* to $list_1$ through $list_n$, then to the cdr of each list, and so on until any of the lists are exhausted. MAPCON concatenates the results using NCONC. Note that MAPCON can easily generate **circular lists**, so beware.

```
(DEFUN MAPCAN (FUNC LST1 LST2)
  ((LAMBDA (LST)
     (LOOP
       ((OR (NULL LST1) (NULL LST2))  LST)
       (SETQ LST (NCONC LST (FUNCALL FUNC (POP LST1) (POP LST2)))) ) )) )

(DEFUN MAPCON (FUNC LST1 LST2)
  ((LAMBDA (LST)
     (LOOP
       ((OR (NULL LST1) (NULL LST2))  LST)
       (SETQ LST (NCONC LST (FUNCALL FUNC LST1 LST2)))
       (POP LST1)
       (POP LST2) ) )) )

$ (MAPCAN '(LAMBDA (N) ((>= N 0) (LIST N))) '(3 -7 4 0 -5 1))
(3 4 0 1)                                          ;Eliminate negative numbers

$ (MAPCON 'REVERSE '(A B C D))                     ;Concatenate reversed tails
(D C B A D C B D C D)
```

5.7.4 Mapping Predicates

The mapping predicates in this subsection successively apply a test function to the elements of one or more lists until a termination criterion is met or the end of any of the lists is reached. The termination criterion is based on the truth value returned by the test function. The mapping predicates return a truth value based on whether or not the end of any of the lists is reached.

SOME [*test*, $list_1$, $list_2$, ..., $list_n$] Function
NOTANY [*test*, $list_1$, $list_2$, ..., $list_n$] Function

(SOME *test* $list_1$ $list_2$ ... $list_n$) applies the predicate *test* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until a test returns a nonNIL value or the end of a list is reached. If a test returns a nonNIL value, SOME returns that value. If the end of a list is reached, SOME returns NIL.

(NOTANY *test* $list_1$ $list_2$ ... $list_n$) is equivalent to (NOT (SOME *test* $list_1$ $list_2$ ... $list_n$)).

```
(DEFUN SOME (TST LST1 LST2)
  (LOOP
    ((OR (NULL LST1) (NULL LST2))  NIL)
    ((FUNCALL TST (POP LST1) (POP LST2))) ) )

(DEFUN NOTANY (TST LST1 LST2)
  (NOT (SOME TST LST1 LST2)) )

$ (SOME 'EQL '(DOG CAT COW) '(COW CAT DOG))                    ;Some pairs are EQL
T

$ (SOME 'PLUSP (LIST 0 -3 (+ 4 -6)))                          ;No element is positive
NIL

$ (NOTANY 'ODDP (LIST 0 (+ 3 3) 7/2))                         ;Not any element is odd
T
```

EVERY [*test*, $list_1$, $list_2$, ..., $list_n$] Function
NOTEVERY [*test*, $list_1$, $list_2$, ..., $list_n$] Function

(EVERY *test* $list_1$ $list_2$ ... $list_n$) applies the predicate *test* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until a test returns NIL or the end of a list is reached. If a test returns NIL, EVERY returns NIL. If the end of a list is reached, it returns T.

(NOTEVERY *test* $list_1$ $list_2$ ... $list_n$) is equivalent to (NOT (EVERY *test* $list_1$ $list_2$ ... $list_n$)).

```
(DEFUN EVERY (TST LST1 LST2)
  (LOOP
    ((OR (NULL LST1) (NULL LST2)))
    ((NOT (FUNCALL TST (POP LST1) (POP LST2)))  NIL) ) )

(DEFUN NOTEVERY (TST LST1 LST2)
  (NOT (EVERY TST LST1 LST2)) )

$ (EVERY 'EQL '(DOG CAT COW) '(DOG CAT PIG))                    ;Not every pair is EQL
NIL

$ (EVERY 'ODDP (LIST 3 5 7 11 13))                               ;Every element is odd
T

$ (NOTEVERY 'EQL '(DOG CAT COW) '(DOG CAT PIG))                 ;Not every element is EQL
T

$ (NOTEVERY 'STRING< '(BILL JACK JOE) '(BUD JIM SUE))
NIL                                          ;Every pair is in alphabetical order
```

REDUCE [*function, list, initial*] Function

If arg_1, arg_2, arg_3 ... and arg_n are the successive elements of *list* and *function* is a two argument function, (REDUCE *function list*) returns

(*function* ... (*function* (*function* arg_1 arg_2) arg_3) ... arg_n)

If *list* is an atom, (REDUCE *function list*) returns (*function*). (REDUCE *function list initial*) is equivalent to (REDUCE *function* (CONS *initial list*)), but requires one less cons.

The reduction is always left-associative. REDUCE makes it possible to apply two argument functions, like UNION and INTERSECTION, to any number of arguments.

```
(DEFUN REDUCE ARGS
  ((OR (NULL ARGS) (NULL (CDR ARGS)))  NIL)
  ((NULL (CDDR ARGS))
    ((ATOM (CADR ARGS))
      (FUNCALL (CAR ARGS)) )
    (reduce-aux (CAR ARGS) (CDADR ARGS) (CAADR ARGS)) )
  (reduce-aux (CAR ARGS) (CADR ARGS) (CADDR ARGS)) )

(DEFUN reduce-aux (FUNC LST RSLT)
  ((ATOM LST)  RSLT)
  (reduce-aux FUNC (CDR LST) (FUNCALL FUNC RSLT (CAR LST))) )

$ (REDUCE 'UNION '((A H E C) (G B A E) (C F A G)))                ;Union of 3 sets
(H B E C F A G)

$ (REDUCE '* '(2 3 5 7))                                  ;Product of a list of numbers
210

$ (REDUCE '* '(2 3 5 7) -2)                                          ;-2 times product
-420
```

5.7.5 Macro Expansion

A **macro call** is a list of the form (*macro* $form_1$ $form_2$ ... $form_n$) where *macro* is the name of a macro. Normally macro expansion occurs automatically when a macro call is pseudo-compiled or run. However, the functions in this subsection allow the user to explicitly expand macro calls.

MACRO-FUNCTION [*symbol*] Library function

If *symbol* is defined as a macro, (MACRO-FUNCTION *symbol*) returns the *symbol*'s macro definition. Otherwise MACRO-FUNCTION returns NIL.

MACRO-FUNCTION is defined in COMMON.LSP.

```
(DEFUN MACRO-FUNCTION (SYM)
  ((MACRO-P SYM)
    (GETD SYM) ) )

$ (MACRO-FUNCTION 'WHEN)                                    ;WHEN's macro definition
(MACRO BODY ((LAMBDA (TEST BODY)
(LIST (QUOTE IF) TEST (CONS (QUOTE PROGN) BODY)))
(CAR (CDR BODY))
(CDR (CDR BODY))))

$ (MACRO-FUNCTION 'FOO)                                     ;FOO is not a macro
NIL
```

MACROEXPAND-1 [*form*] Function
MACROEXPAND [*form*] Function

If *form* is a macro call, (MACROEXPAND-1 *form*) expands *form* (i.e. applies the macro function definition to *form*) and returns the result. Otherwise, MACROEXPAND-1 returns *form*.

If *form* is a macro call, (MACROEXPAND *form*) repeatedly expands *form* until the result of the expansion is not a macro call. MACROEXPAND returns the result of the last expansion.

```
(DEFUN MACROEXPAND-1 (FORM)
  ((ATOM FORM)  FORM)
  ((NOT (SYMBOLP (CAR FORM)))  FORM)
  ((NOT (EQ 'MACRO (GETD (CAR FORM) T)))  FORM)
  (APPLY (CAR FORM) FORM) )

(DEFUN MACROEXPAND (FORM)
  ((ATOM FORM)  FORM)
  ((NOT (SYMBOLP (CAR FORM)))  FORM)
  ((NOT (EQ 'MACRO (GETD (CAR FORM) T)))  FORM)
  (MACROEXPAND (APPLY (CAR FORM) FORM)) )

$ (DEFMACRO EQCAR (OBJ SYM)                          ;Define EQCAR macro
    (LIST 'EQ (LIST 'CAR OBJ) SYM) )
EQCAR

$ (EQCAR '(+ 2 3 5) '+)                               ;Test EQCAR macro
T

$ (MACROEXPAND '(EQCAR '(+ 2 3 5) '+))               ;Macro expansion
(EQ (CAR (QUOTE (+ 2 3 5))) (QUOTE +))
```

MACROEXPAND: T Control variable

If MACROEXPAND is nonNIL, macro calls are expanded as if by the function MACROEXPAND at **compile time** (i.e. when a function is defined using DEFUN or PUTD). If MACROEXPAND is NIL, macro calls are not expanded at compile time.

Although **muLISP** automatically expands macro calls at **run time** (i.e. when a function is being evaluated), it is usually more efficient to expand macros just once at compile time. For macro expansion to occur at compile time, the macro must be defined *before* the function containing the macro call is defined, and MACROEXPAND must be nonNIL.

Since the pseudo-code compiler is non-reentrant, it cannot handle macros calls that define functions. Thus function defining macros must be expanded at run time rather than compile time.

```
$ (SETQ MACROEXPAND)                          ;Turn off compile time expansion

$ (DEFUN IS-SUM (EXPN)                          ;Define IS-SUM using EQCAR
    (EQCAR EXPN '+) )                        ;macro defined on previous page
IS-SUM

$ (GETD IS-SUM)                                    ;Definition of IS-SUM
(LAMBDA (EXPN) (EQCAR EXPN (QUOTE +)))

$ (IS-SUM '(+ 2 3 5))                            ;Verify IS-SUM working
T

$ (SETQ MACROEXPAND T)                        ;Turn on compile time expansion

$ (DEFUN IS-SUM (EXPN)                                ;Redefine IS-SUM
    (EQCAR EXPN '+) )
IS-SUM

$ (GETD IS-SUM)                                ;New definition of IS-SUM
(LAMBDA (EXPN) (EQ (CAR EXPN) (QUOTE +)))

$ (IS-SUM '(+ 2 3 5))                           ;IS-SUM is still working
T                                                      ;and it is faster
```

5.7.6 Memory Management

By recycling and reallocating memory, the memory management system provides programs with a continuous supply of space for new data objects (see Section 4.2). During the collection and reallocation of memory, program execution stops and the blinking cursor temporarily changes to a full block on PC compatible computers.

The primitives in this subsection affect the operation and efficiency of the memory management system.

RECLAIM [ ] Function

(RECLAIM) forces a **garbage collection** to reclaim inactive LISP data structures. If required, a **memory reallocation** is also performed. RECLAIM returns the total number of bytes available in the atom, vector, cons, code, and stack spaces.

Since memory management is automatic, explicit calls on RECLAIM are rarely needed. However calling RECLAIM immediately before a timing test will minimize and standardize the number of garbage collections that occur during the test. Also calling RECLAIM immediately before calling EXECUTE or DOS maximizes the amount of space available for the child process (see Section 5.6.5).

```
(DEFUN RECLAIM ()
  (IF *GCSTAT* (display pre-GC statistics))
  (garbage collect memory in all data spaces)
  (IF *GCSTAT* (display post-GC statistics))
  ((if available free space is not optimally proportioned between data spaces)
    (reallocate free memory among the spaces)
    (IF *GCSTAT* (display post-allocation statistics))
    (return the amount of free space in bytes) )
  (return the amount of free space in bytes) )
```

```
$ (RECLAIM)                                    ;Force a garbage collection
340679                                         ;Return amount of free space
```

GCSTAT: NIL Control variable

If *GCSTAT* is nonNIL, **garbage collection statistics** are displayed on the console before and after all garbage collections and memory reallocations. If *GCSTAT* is NIL, no GC statistics are displayed. Regardless of the value of *GCSTAT*, GC statistics are displayed when a **Memory Full** error occurs to show which space was responsible (see Section 4.5).

The first line of the GC statistics shows the available space just before the garbage collection. The second line shows the available space after the collection. Lines beginning with GC indicate garbage collection statistics; lines beginning with RA indicate data space reallocation statistics. The format for a line of GC statistics is as follows:

GC: *counter atom cons string vector code stack total*

All numbers are in hexadecimal. *counter* is the number of garbage collections since **muLISP** began execution. The remaining numbers are the amount of bytes free in the atom, cons, string, vector, code, and stack spaces (see Section 4.2). *total* is the total number of bytes free.

Two bytes are required for each **muLISP** pointer cell in the atom, cons, code, and stack spaces. One byte is required for each character in a print name and two bytes are required to store the length of a print name. Number vectors use only as many two byte words as necessary to store the number in binary.

The GC statistics are heavily dependent on the implementation of **muLISP** being run and should normally be of no concern to users. However, they are useful for determining the cause of **thrashing**. Thrashing occurs when the system is forced to spend an inordinate amount of time garbage collecting for a very small return of free data space.

```
$ (SETQ *GCSTAT* T)                                    ;Turn on GC statistics flag
T

$ (LOOP (OBLIST))                                      ;Create lots of conses

GC: 0001  0A7A0  00000  0EC9F  0FFFE  0D120  02AB6  39103
GC: 0001  0A7A8  19F00  0ECA1  0FFFE  0D120  02AB6  52F1D
RA: 0001  00228  1F0C0  0ECA1  0FFFE  0D3A0  02836  4DB5D

GC: 0002  00228  00000  0ECA1  0FFFE  0D3A0  02836  2EA9D
GC: 0002  00228  1F218  0ECA1  0FFFE  0D3A0  02836  4DCB5

GC: 0003  00228  00000  0ECA1  0FFFE  0D3A0  02836  2EA9D
GC: 0003  00228  1F218  0ECA1  0FFFE  0D3A0  02836  4DCB5
```

This "endless" loop and the display of GC statistics will continue until a console interrupt is initiated by pressing the **Esc** key.

The three lines of GC statistics for garbage collection number 1 indicate that a memory reallocation occurred. More memory was allocated to the cons space because of the heavy demand by OBLIST for new conses. Subsequent garbage collections did not perform reallocations because the data space sizes were already optimized. However, if the program changed and started using the atom space, for example, another reallocation would occur to expand the size of that region at the expense of the others.

The example shown above was done on an 80286 based computer with 640K (kilobytes) of memory. The numbers will vary greatly depending on the amount of memory available to **muLISP**.

GCHOOK: NIL — Control variable

If *GCHOOK* is the name of a function, the function is called *after* each garbage collection. If *GCHOOK* is NIL, no call is made. Otherwise, an Undefined Function error break occurs.

By assigning to *GCHOOK* a function that monitors the amount of free space after garbage collections, programs may be able to automatically prevent thrashing and/or Memory Full errors by the release of unneeded data when the amount of free space becomes dangerously low.

ZAP-STRING [*symbol*] Function

(ZAP-STRING *symbol*) frees the bytes required to store *symbol*'s print name and makes them available for reuse. Thereafter, *symbol* will print as || (i.e. the null string). ZAP-STRING returns T if *symbol* is a symbol, otherwise it returns NIL.

Print names of symbols are only used by the communication primitives (see Section 5.8) and by the print name primitives (see Section 5.5). The space available to store print names is limited due to segmentation constraints imposed by the Intel 8086 microprocessor (see Section 4.2). If a symbol does not need to be printed or read again, ZAP-STRING can free the space required to store the symbol's print name.

For instance, "zapping" the print names of the internal functions of a sizable program can save a considerable amount of string space. The following example saves 16 bytes of string space by zapping the function SCROLL-UP-SCREEN defined in the file EDIT.LSP of the **muLISP** program development system (see Section 7.2):

```
$ (ZAP-STRING 'SCROLL-UP-SCREEN)
T
```

FREE-LIST: NIL Variable

The constructor functions are required to create new conses. If *FREE-LIST* is an atom, space for the new conses is taken from the **cons space** (see Section 4.2). If previous consing has exhausted the cons space, a garbage collection is automatically performed to reclaim the space used by conses that are no longer accessible.

However, if *FREE-LIST* is a cons, that cons is used when a new cons is required by a constructor function. By nconcing no longer needed lists to *FREE-LIST*, application programs can reduce or eliminate the occurrence of garbage collections.

Great care must be taken to insure that conses nconced to *FREE-LIST* are not going to be referenced again. We recommend that *FREE-LIST* not be incorporated into an application program until the program has been thoroughly debugged.

The following example adds 6 (*not* 9) conses to *FREE-LIST*:

```
$ (SETQ JUNK '(A B (C D E) 3 4 5))
(A B (C D E) 3 4 5)

$ (SETQ *FREE-LIST* (NCONC JUNK *FREE-LIST*))
(A B (C D E) 3 4 5)
```

ALLOCATE [*n*] — Function

If *n* is a positive integer, (ALLOCATE *n*) frees *n* bytes of memory in the 8086 code segment following the **muLISP** machine code and returns the base offset address of the newly allocated space. If there is insufficient room to free *n* bytes, ALLOCATE returns NIL.

(ALLOCATE) returns the current base offset address of the beginning of free space in the 8086 code segment following the **muLISP** machine code.

ALLOCATE is primarily intended for compilers that need space to store the machine code they generate.

```
(DEFUN ALLOCATE (N)
  ((PLUSP N)
    ((if there is sufficient free memory)
      (free N bytes of memory after machine code)
      (return base offset address of free space) ) )
    (return base offset address of free space) )

$ (ALLOCATE 100)                                    ;Free 100 bytes of memory
39120                                          ;Free memory begins at offset 39120

$ (ALLOCATE)                                      ;Return start of next free space
39232
```

NEW-CODE-SPACE [ ] Function

(NEW-CODE-SPACE) creates a new code space for the storage of function definitions (see Section 4.2) and returns T. There can be as many as six code spaces. If six code spaces already exist or there is insufficient memory for a new space, NEW-CODE-SPACE returns NIL.

Provided there is sufficient computer memory, each code space can store about 55K of pseudo-code. This allows **muLISP** to take advantage of the entire 640K MS-DOS work space and makes it possible to load well over 25000 lines (about 500 pages) of LISP source code. The judicious use of NEW-CODE-SPACE can also avoid Memory Full errors (see Section 4.5) that are caused by the exhaustion of the atom space (see Section 4.2).

muLISP automatically creates a new space when function definitions have filled the current code space. If a function in one code space calls a function in another space, a slight speed penalty must be paid while **muLISP** switches to the new space. To reduce the effect of this code space "fault", large programs (i.e. those which cannot be stored in a single code space) can be divided into modules that operate relatively independently of one another so that each module can be loaded into a separate code space.

Thus to make maximum use of available memory, simply break up the source file for large programs into modules each less than 80 pages in length. Then load the first module, call NEW-CODE-SPACE, load the second module, call NEW-CODE-SPACE again, etc.

If only a single code space is being used and full condensing has not been in effect, the space used to store the original definition of a function being redefined is automatically reclaimed (see *CONDENSE* in Section 5.10). However, if additional code spaces have been created (either automatically or by NEW-CODE-SPACE) and a function is redefined, the space for the original definition is *not* reclaimed.

```
$ (NEW-CODE-SPACE)                          ;Create a new code space
T
```

5.8 Communication

The primitives in this section enable programs to communicate with the console, the printer, data files, and other system hardware.

5.8.1 Input Functions

The input functions read and return data from an input source, either the console or an open input file. If the *source* argument to an input function is NIL or omitted, input is received from the **standard input source** (see Section 5.9.1). If the *source* argument is T, input is received from the console. If the *source* argument is an open input file, input is received from the file.

If a file is the input source, the following **end-of-file error processing** occurs when an input function attempts to read past the end of the file: If the *eof-value-p* argument to the function is NIL or omitted, an End-Of-File error break is generated (see Section 4.5). If the *eof-value-p* is nonNIL, the *eof-value* argument to the function is returned.

If the console is the input source, the input functions *other* than READ-BYTE invoke the **muLISP line editor** when all the characters have been read from the current line. Characters are echoed on the console screen as they are typed. Until the **Enter** key is pressed, the line editing commands can be used to edit the line (see Section 2.3). This **line edit mode** of console input provides a uniform interface suitable for most standard applications.

If the console is the input source, the input function READ-BYTE returns the ASCII code of keyboard characters as soon as they are pressed. Characters are *not* echoed on the console screen. This **raw input mode** of console input is useful for applications that need to process characters as soon as they are entered by the user.

READ [*source, eof-value-p, eof-value*] Function

READ reads one complete symbolic expression from the specified input source and returns the equivalent object. Well-formed expressions using either the list notation, dot notation, or a combination of both are acceptable input for READ (see Section 4.1.3).

Whitespace characters serve only to delimit tokens being read by READ. Initially, the whitespace characters are the SPACE, RETURN, LINEFEED, FORMFEED, and TAB characters. Use COPY-CHAR-TYPE to change the whitespace characters.

Escape characters defeat the syntax properties of whitespace, macro, and escape characters so that they can be included as part of a token being read by READ. **Single escape characters** delimit the character immediately following them. **Multiple escape characters** delimit the characters between them and the next multiple escape character. Single escape characters can be used to include single and multiple escape characters in tokens. Initially, the single escape character is \ (the BACKSLASH character). Initially, the multiple escape character is | (the VERTICAL-BAR character). The initial escape characters can be changed by modifying the **Default Read Table** in the **muLISP** base page (see Appendix H).

Macro characters invoke their macro definition function when the character is read by READ (see Section 5.8.2).

```
$ (READ)   (A (B . C) D)                                    ;Read a list
(A (B . C) D)

$ (READ)   |This symbol is delimited by multiple escape characters|
|This symbol is delimited by multiple escape characters|

$ (READ)   '(+ 2 3)                                  ;Read a macro character
(QUOTE (+ 2 3))

$ (READ)   \"                                        ;Read the double quote character
\"
```

READ-ATOM [*source, eof-value-p, eof-value*] Function
BREAK-CHARS [*list, flag*] Function

READ-ATOM reads a token from the specified input source and returns the corresponding atom. A **token** is a string of characters delimited by either whitespace or break characters.

Whitespace characters serve only to delimit tokens being read by READ-ATOM and are *not* returned as atoms. Initially, the whitespace characters are the SPACE, RETURN, LINEFEED, FORMFEED, and TAB characters. READ-ATOM and READ use the same whitespace characters. Use COPY-CHAR-TYPE to change the whitespace characters.

Break characters read by READ-ATOM are returned as symbols whose print names consist of a single character. Like whitespace, break characters also delimit tokens. Initially, the break characters are:

```
   !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  :
   <  =  >  ?  @  [  \  ]  ^  _  `  {  |  }  ~
```

READ-ATOM recognizes both single and multiple escape characters as a means of including whitespace and/or break characters as part of a token (see READ).

If it reads a token beginning with a *decimal* digit less than the input radix base, READ-ATOM returns the corresponding number. Such tokens are terminated by any character not a digit in the input radix base (see *READ-BASE* in Section 5.8.3). If it reads a token beginning with other than a decimal digit, READ-ATOM returns the symbol which has the token as its print name.

(BREAK-CHARS *list*) makes the characters in *list* the set of break characters accepted by READ-ATOM and returns NIL. If *flag* is nonNIL, (BREAK-CHARS *list flag*) adds the characters in *list* to the current set of break characters and returns NIL. (BREAK-CHARS) returns a list of the current set of break characters.

```
$ (READ-ATOM)   -456                                    ;Read and return a hyphen
-                                                         ;456 remains for input

$ (READ-ATOM)   123CAT                                    ;Read and return 123
123                                                       ;CAT remains for input

$ (DEFUN UNBREAK-CHAR (CHAR)                          ;Makes CHAR a nonbreak character
    (BREAK-CHARS (DELETE CHAR (BREAK-CHARS))) )

$ (UNBREAK-CHAR '-')                                      ;Make - a nonbreak char
NIL

$ (READ-ATOM) TOP-LEVEL                                 ;Read TOP-LEVEL as one symbol
TOP-LEVEL
```

READ-LINE [*source, eof-value-p, eof-value*] Function
READ-RECORD [*n, source, eof-value-p, eof-value*] Function
READ-CHAR [*source, eof-value-p, eof-value*] Function
READ-BYTE [*source, eof-value-p, eof-value*] Function

READ-LINE reads characters from the specified input source until the RETURN character is encountered, and returns the symbol whose print name consists of all the characters read except for the RETURN. If there is a LINEFEED immediately following the RETURN, it is discarded along with the RETURN. READ-LINE does *not* discard blanks, recognize comments, or convert lower-case letters to upper-case.

READ-RECORD reads the next *n* characters from the specified input source, and returns the symbol whose print name consists of those characters. READ-RECORD always returns a symbol, even if the characters it reads are digits.

READ-CHAR reads the next character from the specified input source and returns it as the symbol whose print name consists of that character. READ-CHAR always returns a symbol, even if the character it reads is a digit.

READ-BYTE reads the next byte from the specified input source and returns it in the form of an integer between 0 and 255 inclusive. If READ-BYTE attempts to read past the *physical* end of a file, end-of-file error processing occurs. However if READ-BYTE reads the EOF character (the *logical* end-of-file character), *no* end-of-file error processing occurs.

```
$ (READ-LINE) One small step for man                    ;Read a line
| One small step for man|                              ;Return a symbol

$ (READ-RECORD 10) One small step for man              ;Read 10 characters
| One small|                                           ;Return a symbol

$ (READ-CHAR)a                                          ;Read a character
\a                                                      ;Return a symbol

$ (READ-BYTE)a                                          ;Read a character
97                                                    ;Return the ASCII code
```

UNREAD-CHAR [*source*] Function
PEEK-CHAR [*peek-type*, *source*, *eof-value-p*, *eof-value*] Function

UNREAD-CHAR restores the last character or byte read from the specified input source to the head of the input source, and returns NIL. Since only the most recently read character or byte can be restored, calling UNREAD-CHAR a second time on the same input source without an intervening read operation has no additional effect.

(PEEK-CHAR) reads the next character from the specified input source. (PEEK-CHAR T) reads characters from the specified input source until a non-whitespace character is encountered. If *character* is a symbol, (PEEK-CHAR *character*) reads characters from the specified input source until a character equal to the first character of *character*'s print name is encountered. In all the cases, PEEK-CHAR restores the last character read from the specified input source to the head of the input source, and returns the symbol whose print name consists of that character.

```
$ (PROG1 (READ-CHAR) (UNREAD-CHAR))COMPUTER       ;Read "C", unread "C", and return "C"
C

$ COMPUTER                          ;This is read and displayed by the read-eval-print loop

$ (PEEK-CHAR)COMPUTER                             ;Read "C", unread "C", and return "C"
C

$ COMPUTER                                ;Read and displayed by the read-eval-print loop

$ (PEEK-CHAR T)      COMPUTER
C                                         ;Read till non-whitespace character encountered

$ COMPUTER                          ;This is read and displayed by the read-eval-print loop

$ (PEEK-CHAR 'M) COMPUTER                                    ;Read till "M" encountered
M

$ MPUTER                            ;This is read and displayed by the read-eval-print loop
```

CLEAR-INPUT [*source*] Function
LISTEN [*source*] Function

If the console is the input source, CLEAR-INPUT clears the console input buffer and returns NIL. If a file is the input source, CLEAR-INPUT has no effect and just returns NIL. After a program detects an error and before querying the user on how to proceed, CLEAR-INPUT is useful for emptying the console input buffer of now extraneous input.

If the console is the input source and a key has been pressed, LISTEN returns T, otherwise it returns NIL. If a file is the input source and a character is available for input (i.e. the read position is *not* at the end of the file), LISTEN returns T, otherwise it returns NIL.

The example function BEEP-READ-BYTE clears the console input buffer and periodically sounds a beep until the user presses a key, at which time the ASCII code for the key is returned. BEEP-READ-BYTE uses CLEAR-INPUT to clear the console input buffer. Then it uses LISTEN to determine if the user has pressed a key.

```
$ (DEFUN BEEP-READ-BYTE (
     CTR )                                          ;Make CTR a local variable
   (CLEAR-INPUT T)                                  ;Empty console input buffer
   (SETQ CTR 0)                                     ;Initialize CTR
   (LOOP
     ((LISTEN T)                                    ;Key pressed?
       (READ-BYTE T) )                              ;Read key
     (TONE NIL 100)                                 ;Pause 1/10 second
     (INCQ CTR)                                     ;Increment CTR
     (IF (= CTR 50)                                 ;5 seconds elapsed?
         (PROGN (SETQ CTR 0)                        ;Reinitialize CTR
                (WRITE-BYTE 7 NIL T))) ) )          ;Beep console

$ (BEEP-READ-BYTE)                                 ;Waits until a key is pressed
```

READ-FROM-STRING [*string, eof-value-p, eof-value*] Function
STRING-INDEX Variable
READ-STRINGS Variable

If *strings* is a nonempty list of symbols, (READ-FROM-STRING *strings*) reads one complete symbolic expression from the symbols in *strings*, sets *READ-STRINGS* to the tail of *strings* beginning with the last symbol read, sets *STRING-INDEX* to 1 plus the index of the last character read, and returns the object equivalent to the expression read. A RETURN character is inserted in the input stream between each string read by READ-FROM-STRING.

Note that the first element of the list *READ-STRINGS* is the *entire* print name string being read, *not* just the part of the string not yet read. To extract just the part not yet read, use the command (SUBSTRING (CAR *READ-STRINGS*) *STRING-INDEX*).

If *string* is a symbol, (READ-FROM-STRING *string*) is equivalent to (READ-FROM-STRING (LIST *string*)).

READ-FROM-STRING accepts well-formed expressions expressed using either list notation, dot notation, or a combination of both (see Section 4.1.3). READ-FROM-STRING processes whitespace, escape, and macro characters the same way READ processes them. Also, READ-FROM-STRING processes attempts to read past the end of a string the same way READ processes attempts to read past the end of a file.

```
$ (READ-FROM-STRING "(1 2 3)")
(1 2 3)

$ *STRING-INDEX*
7

$ (READ-FROM-STRING "apple orange")
APPLE

$ *STRING-INDEX*
5

$ (READ-FROM-STRING '("(1 2"  "3 4)"  "5 6"))
(1 2 3 4)

$ *READ-STRINGS*
(|3 4)| |5 6|)
```

PARSE-INTEGER [*string, radix*] Library function
STRING-INDEX Variable

If *radix* is an integer between 2 and 36 inclusive, (PARSE-INTEGER *string radix*) skips any leading whitespace, attempts to read and parse an integer in base *radix* from *string*, sets *STRING-INDEX* to 1 plus the index of the last valid digit character it read, and returns the value of the integer it parsed. *radix* defaults to ten.

PARSE-INTEGER accepts integers consisting of an optional plus or minus sign followed by one or more digits in base *radix*. It stops reading and returns the value of the integer read when an invalid digit or the end of *string* is encountered. If the first non-whitespace characters of *string* do not constitute a valid integer, PARSE-INTEGER returns NIL.

PARSE-INTEGER is defined in COMMON.LSP.

```
$ (PARSE-INTEGER "123")
123

$ *STRING-INDEX*
3

$ (PARSE-INTEGER "  -0054321+123")
-54321

$ *STRING-INDEX*
10
```

5.8.2 Read Macro Characters

Macro characters invoke their macro function definitions when they are encountered by the functions READ and READ-FROM-STRING. In addition to making it possible to abbreviate frequently used code segments, macro characters can alter the syntax used in source files to better suit the application at hand.

Macro characters can be categorized as being either **terminating, non-terminating,** or **comment.** Terminating and non-terminating macro characters insert exactly one object into the input stream when they are encountered. Comment macro characters insert no object into the input stream. Comment and terminating macro characters will terminate a token being read by READ. Non-terminating macro characters do *not* terminate a token being read and hence must be preceded by at least one whitespace character in order to be recognized as a macro character.

Several of the most commonly used macro characters are pre-defined in **muLISP.** The function COPY-CHAR-TYPE can be used to assign the definition of these macro characters to other characters. The function SET-MACRO-CHAR can be used to define new macro characters.

COPY-CHAR-TYPE [$char_1$, $char_2$, *flag*] Function

If *flag* is nonNIL, (COPY-CHAR-TYPE $char_1$ $char_2$ *flag*) copies the character type of $char_2$ in the current readtable to $char_1$. If *flag* is NIL or omitted, COPY-CHAR-TYPE copies the character type of $char_2$ in the default readtable to $char_1$. If $char_2$ is a macro character, a pointer to its macro definition is also copied to $char_1$. If successful, COPY-CHAR-TYPE returns T, otherwise it returns NIL.

```
$ (COPY-CHAR-TYPE '\~ '\;)                    ;Make ~ a comment char
T
```

SET-MACRO-CHAR [*char, definition, flag*] Function
GET-MACRO-CHAR [*char, flag*] Function
COMMENT Keyword

(SET-MACRO-CHAR *char definition*) uses *definition* to establish *char* as a terminating macro character. (SET-MACRO-CHAR *char definition* COMMENT) establishes *char* as a comment macro character. (SET-MACRO-CHAR *char definition* T) establishes *char* as a non-terminating macro character.

(GET-MACRO-CHAR *char*) returns the macro character definition associated with *char*. If *flag* is nonNIL, (GET-MACRO-CHAR *char flag*) returns LAMBDA if *char* is a macro character. In either case, GET-MACRO-CHAR returns NIL if *char* is not a macro character.

The following example shows how to define the percent sign (%) as a comment macro that recognizes text enclosed in matching percent signs as a comment. Unlike comments delimited by a semicolon, comments delimited by percent signs can be more than a single line and can have noncommented expressions to their right as well as left:

```
$ (SET-MACRO-CHAR '\%                                    ;Define % comment macro
    '(LAMBDA ()
      (PEEK-CHAR '\%)
      (READ-CHAR) )
    'COMMENT)

$ '(DOG CAT % This is a comment. % COW)
(DOG CAT COW)
```

((left parenthesis) Terminating macro character

The left parenthesis macro reads expressions until a matching right parenthesis, right square bracket, or dot is encountered. If a right parenthesis is encountered, the macro returns a list of the expressions read. If a right square bracket is encountered, the macro "unreads" the bracket so that it can be read again and the macro returns a list of the expressions read. This makes the right square bracket a **super-parenthesis** that closes all currently unmatched left parentheses.

If a dot is encountered, the macro reads one more expression and checks to ensure that the next non-whitespace character read is either a right parenthesis or a right square bracket. If so, the macro returns a list of the expressions read before the dot was encountered, nconced with the expression read after the dot. If not, a Syntax Error break occurs.

) (right parenthesis) Terminating macro character
] (right square bracket) Terminating macro character

The left parenthesis macro recognizes the right parenthesis and the right square bracket as list terminators. The right parenthesis macro and the right square bracket macro generate a Syntax Error break if too many right parentheses or right square brackets are encountered by READ.

, (comma) Terminating macro character

The comma macro generates a Syntax Error break if a comma is encountered directly by READ.

' (single quote) Terminating macro character

The single quote mark macro reads in the expression following the mark and returns a two element list consisting of the symbol QUOTE and the expression just read. The single quote simplifies the entry of quoted expressions.

; (semicolon) Comment macro character

The semicolon macro reads in characters until the RETURN character is encountered. Since no object is inserted into the input stream, this makes semicolon a single-line comment delimiter.

" (double quote) Terminating macro character

The double quote macro reads characters from the input stream until a matching double quote character is encountered and returns the symbol whose print name consists of the characters read. Single escape characters (normally \) can be used to include double quote characters and single escape characters in the string.

` (backquote) Library terminating macro character

When defining macros using DEFMACRO (see Section 5.3.3), executable code must be created for evaluation at runtime. This code can be created directly using constructor functions such as CONS, APPEND, LIST, and LIST*. However if the code to be generated is at all complex, macros defined in this way are difficult to understand and debug.

The **backquote macro** character facility can greatly simplify macro definitions. It lets you create code or other complex data structures by giving a "template" for the desired data structure. Also you can specify variables in the template so that they will be filled in by their values.

An expression preceded by a comma is a variable whose value must be inserted in the template at that position. An expression preceded by a comma and an at-sign (@) is a variable whose value must be *spliced* into the list in the template in which the expression occurs. More detailed explanations of the backquote macro can be found the many books describing Common LISP (see Section 10.3).

The following example illustrates the code created by the backquote macro and the constructor functions necessary to produce directly the equivalent code:

```
$ (SETQ X 7 Y '(2 3 5))
(2 3 5)

$ `(IF (NUMBERP ,X) (+ ,X ,@Y))
(IF (NUMBERP 7) (+ 7 2 3 5))

$ (LIST 'IF (LIST 'NUMBERP X) (LIST* '+ X Y))
(IF (NUMBERP 7) (+ 7 2 3 5))
```

5.8.3 Input Control Variables

The input control variables extend the flexibility of the input functions by making several input options available to the user. Most operate as switches that are *off* when their value is NIL and *on* when their value is nonNIL.

INPUT-ECHO: NIL Control variable

If *INPUT-ECHO* is nonNIL, the characters read from files are echoed to the standard output sink. If *INPUT-ECHO* is NIL, characters are not echoed to the standard output sink. Since comments are also echoed, English language text within a comment can be displayed conveniently without having to actually process the text.

READ-BASE: 10 Control variable

If *READ-BASE* is an integer between 2 and 36 inclusive, READ, READ-FROM-STRING, and READ-ATOM use that integer as the **input radix base** when reading numbers. Otherwise, these functions use radix base ten when reading numbers.

The initial value of *READ-BASE* is 10. *PRINT-BASE* controls the output radix base used when printing numbers.

READ-UPCASE: T Control variable

If *READ-UPCASE* is nonNIL, READ, READ-FROM-STRING, and READ-ATOM convert lower-case letters to upper-case as they are read unless the letters are part of a string enclosed in double quotes, or the letters are delimited by single or multiple escape characters. If *READ-UPCASE* is NIL, these functions do not convert lower-case letters to upper-case.

Regardless of the value of *READ-UPCASE*, print names already in the system containing lower-case letters are not changed.

TAB-EXPAND: NIL Control variable

If *TAB-EXPAND* is nonNIL, the TAB character is expanded to between one and eight spaces when it is read by an input function other than READ-BYTE. If *TAB-EXPAND* is NIL, the TAB character is not expanded.

Regardless of the value of *TAB-EXPAND*, READ-BYTE returns 9 when the TAB character is read. *BLANK-COMPRESS* controls the converse operation of compressing of BLANK characters to a single TAB character (see Section 5.8.5).

IGNORE-CASE: NIL Control variable

When a print name read by either the function READ, READ-FROM-STRING, or READ-ATOM is being interned, **muLISP** first attempts to find and return a symbol in the current oblist that has the same print name.

If no such symbol can be found, and *IGNORE-CASE* is NIL *or* a file is the input source, a symbol having the new print name is added to the oblist. If *IGNORE-CASE* is nonNIL *and* the console is the input source, **muLISP** ignores the case of letters while attempting to find a symbol in the oblist that has the same print name as the print name being interned. If a matching symbol is still not found, a symbol having the new print name is added to the oblist. *IGNORE-CASE* only affects the functions READ, READ-FROM-STRING, and READ-ATOM.

5.8.4 Output Functions

The output functions format and send data to an open output sink: either the console, an open output file, or an open output device (such as LPT1 for parallel printer #1 or COM1 for serial printer #1). Use the file system functions to open an output file or device (see Section 5.9.2).

If the *sink* argument to an output function is NIL or omitted, output is sent to the **standard output sink** (see Section 5.9.2). If the *sink* argument is T, output is sent to the console. If the *sink* argument is an open output file or device, output is sent to that file or device.

If a file is the output sink, the following **disk full error** processing occurs when an output function attempts to write data to a disk file and there is insufficient space on the disk to store the data: A throw of NIL is made to the label Disk Full. If this throw is not caught by an application program, the top-level loop (see Section 5.10.1) catches the throw and generates a Disk Full error break (see Section 4.5).

WRITE [*object*, *sink*] Function

(WRITE *object sink*) sends the printed representation of *object* to *sink* and returns *object*. WRITE uses the value of the output control variables (see Section 5.8.5) to determine how atoms in the object are represented.

Symbols are represented by the string of characters in their print name. *PRINT-ESCAPE* controls the use of escape characters around print names containing whitespace, macro, or escape characters. *PRINT-DOWNCASE* controls the conversion of upper-case letters to lower-case.

Numbers are represented by the string of numerals equivalent to the number. *PRINT-BASE* controls the output radix base used to represent numbers. *PRINT-POINT* controls the notation and the maximum number of digits used to represent ratios.

Conses are represented using **list notation** where possible and **dot notation** where necessary (see Section 4.1.3).

```
(DEFUN WRITE (OBJ SINK)
  (send the printed representation of OBJ to SINK)
  OBJ )

$ (WRITE '(A . (B . C)))
(A B . C)
```

PRIN1 [*object*, *sink*] Function
PRINT [*object*, *sink*] Function
PRINC [*object*, *sink*] Function
PPRINT [*object*, *sink*] Library function

Using escape characters as necessary, (PRIN1 *object sink*) sends the printed representation of *object* to *sink* and returns *object*. PRIN1 is equivalent to WRITE with *PRINT-ESCAPE* temporarily bound to T. PRIN1 produces output suitable for input by READ.

Using escape characters as necessary, (PRINT *object sink*) sends the printed representation of *object* to *sink*, then it sends the NEWLINE character pair (see TERPRI) to *sink*, and finally it returns *object*. PRINT is equivalent to calling PRIN1 and then TERPRI.

Using *no* escape characters, (PRINC *object sink*) sends the printed representation of *object* to *sink* and returns *object*. PRINC is equivalent to WRITE with *PRINT-ESCAPE* temporarily bound to NIL. PRINC produces output more suitable for human readers.

Using indentation and newlines to make the result easier to read, (PPRINT *object sink*) sends the printed representation of *object* to *sink* and returns NIL. PPRINT is defined in COMMON.LSP.

```
(DEFUN PRIN1 (OBJ SINK
    *PRINT-ESCAPE* )
  (SETQ *PRINT-ESCAPE* T)
  (WRITE OBJ SINK) )

(DEFUN PRINT (OBJ SINK)
  (PRIN1 OBJ SINK)
  (TERPRI 1 SINK)
  OBJ )

(DEFUN PRINC (OBJ SINK
    *PRINT-ESCAPE* )
  (SETQ *PRINT-ESCAPE* NIL)
  (WRITE OBJ SINK) )

$ (PRINT '(A B C))                          ;Print and return list
(A B C)
(A B C)

$ (PRINC |The quick brown fox|)             ;Print and return phrase
The quick brown fox|The quick brown fox|
```

WRITE-STRING [*symbol*, *sink*] Function
WRITE-LINE [*symbol*, *sink*] Function

If *symbol* is a symbol, (WRITE-STRING *symbol sink*) sends the characters in *symbol*'s print name to *sink* and returns *symbol*. If its first argument is not a symbol, WRITE-STRING returns NIL.

If *symbol* is a symbol, (WRITE-LINE *symbol sink*) sends the characters in *symbol*'s print name to *sink*, then it sends the NEWLINE character pair (see TERPRI) to *sink*, and finally it returns *symbol*. If its first argument is not a symbol, WRITE-LINE returns NIL.

Both WRITE-STRING and WRITE-LINE ignore the value of the control variables *PRINT-ESCAPE* and *PRINT-DOWNCASE* and output the characters exactly as they occur in *symbol*'s print name.

```
(DEFUN WRITE-STRING (SYM SINK)
  ((SYMBOLP SYM)
    (send OBJ'S printed representation to SINK)
    SYM ) )

(DEFUN WRITE-LINE (SYM SINK)
  ((SYMBOLP SYM)
    (WRITE-STRING SYM SINK)
    (TERPRI)
    SYM ) )

$ (WRITE-STRING |muLISP|)
muLISP|muLISP|                                  ;Write and return "muLISP"

$ (WRITE-LINE |muLISP|)
muLISP                                      ;Write "muLISP" and a newline
|muLISP|                                                ;Return "muLISP"
```

SPACES [*n*, *sink*] Function
TERPRI [*n*, *sink*] Function
FRESH-LINE [*sink*] Function

If *n* is zero or a positive integer, (SPACES *n sink*) sends *n* SPACE characters to *sink* and returns the number of characters sent since the last NEWLINE character pair was sent. *n* defaults to 0.

If *n* is zero or a positive integer, (TERPRI *n sink*) sends *n* NEWLINE character pairs to *sink* and returns NIL. *n* defaults to 1. A NEWLINE character pair consists of a RETURN character followed by a LINEFEED character.

If a NEWLINE character pair was *not* the last thing sent to *sink*, (FRESH-LINE *sink*) sends a NEWLINE character pair to *sink* and returns T. Otherwise, FRESH-LINE has no effect and just returns NIL.

```
(DEFUN SPACES (N SINK)
  ((AND (INTEGERP N) (>= N 0))
    (LOOP
      ((ZEROP N)
        (characters sent since last newline) )
      (WRITE-BYTE 32 1 SINK)
      (DECQ N) ) )
  (SPACES 0 SINK) )
(DEFUN TERPRI (N SINK)
  ((AND (INTEGERP N) (>= N 0))
    (LOOP
      ((ZEROP N)  NIL)
      (WRITE-BYTE 13 1 SINK)
      (WRITE-BYTE 10 1 SINK)
      (DECQ N) ) )
  (TERPRI 1 SINK) )

(DEFUN FRESH-LINE (SINK)
  ((ZEROP (SPACES SINK))  NIL)
  (TERPRI SINK)
  T )

$ (DEFUN TAB (N)                                    ;Define a tab function
    (SPACES (- N (SPACES))) )
TAB

$ (PROGN (PRIN1 'MARS) (TERPRI 2) 'VENUS)
MARS                                                ;Print MARS
                                                    ;Move down 2 lines
VENUS                                               ;Print VENUS
```

WRITE-BYTE [*n*, *m*, *sink*] Function

If *n* is an integer between 0 and 255 inclusive and *m* is zero or a positive integer, (WRITE-BYTE *n m sink*) writes *m* bytes of *n* to *sink* and returns *n*. *m* defaults to 1. If *n* is not within range, WRITE-BYTE returns NIL.

WRITE-BYTE is useful for displaying on the console screen most of the 256 possible **ASCII characters**, including those with codes greater than 127. Codes 8 (BACKSPACE), 10 (LINEFEED), and 13 (RETURN) move the cursor rather than display a character. Code 7 (BELL) makes a beep. If running on a PC compatible computer, Appendix C lists the decimal and hexadecimal ASCII code for each PC screen character.

```
(DEFUN WRITE-BYTE (N M SINK)
  ((AND (INTEGER N) (<= 0 N 255))
    ((AND (INTEGER M) (>= M 0))
      (LOOP
        ((ZEROP M)  N)
        (write N to SINK)
        (DECQ M) ) )
    (WRITE-BYTE N 1 SINK) ) )

$ (WRITE-BYTE 65)                                    ;Write "A" and return 65
A65

$ (WRITE-BYTE 65 10)                                 ;Write 10 "A"s and return 65
AAAAAAAAAA65

$ (WRITE-BYTE 10)                                    ;Write a linefeed and return 10

10
```

5.8.5 Output Control Variables

The output control variables enhance the control programs have over character output. Most operate as switches that are *off* when their value is NIL and *on* when their value is nonNIL.

AUTO-NEWLINE: T Control variable

If *AUTO-NEWLINE* is nonNIL and a file is the output sink, NEWLINE character pairs are automatically sent to the file when necessary to enforce the line length of the file. If *AUTO-NEWLINE* is nonNIL and the console is the output sink, NEWLINE character pairs are automatically sent to the console after a character is displayed in the right-most column of the current window (see Section 5.8.6).

If *AUTO-NEWLINE* is NIL and a file is the output sink, the line length of the file is *not* enforced. If *AUTO-NEWLINE* is nonNIL and the console is the output sink, when the cursor reaches the right end of the line, the cursor simply remains there while new characters overlay previous ones.

BLANK-COMPRESS: NIL Control variable

If *BLANK-COMPRESS* is nonNIL, when two to eight BLANK characters ending on a **tab stop** are encountered by an output function other than WRITE-BYTE, they are compressed to a single TAB character before being sent to the output sink. If *BLANK-COMPRESS* is NIL, the BLANK characters are not compressed.

Regardless of the value of *BLANK-COMPRESS*, (WRITE-BYTE 32) sends a BLANK character to the output sink. *TAB-EXPAND* controls the converse operation of expanding a TAB character to multiple SPACE characters (see Section 5.8.3).

OUTPUT-ECHO: NIL Control variable

If *OUTPUT-ECHO* is nonNIL and a disk file is the output sink, characters sent to the file are also sent to the console. If *OUTPUT-ECHO* is NIL, characters are not echoed to the console.

PRINTER-ECHO: NIL Control variable

If *PRINTER-ECHO* is nonNIL, characters sent to the console are also sent to the system printer. If *PRINTER-ECHO* is NIL, characters are not echoed to the printer.

Setting *PRINTER-ECHO* to true is a convenient means of producing a printed record of a series of read-eval-print loop interactions (see Section 2.2).

PRINT-DOWNCASE: NIL Control variable

If *PRINT-DOWNCASE* is nonNIL, WRITE, PRIN1, PRINT, and PRINC convert upper-case letters to lower-case as they are sent to an output sink. If *PRINT-DOWNCASE* is NIL, these functions do not convert upper-case letters to lower-case. **Case conversion** is only done as the characters are being output and does not affect the actual print name stored in memory.

```
$ (SETQ *PRINT-DOWNCASE* T)                    ;Turn lower-case conversion on
T

$ 'muLISP                                      ;Displays in lower case
mulisp

$ (SETQ *PRINT-DOWNCASE* NIL)                  ;Turn lower-case conversion off
NIL

$ 'muLISP                                      ;Displays in upper case
MULISP
```

PRINT-ESCAPE: T Control variable

If *PRINT-ESCAPE* is nonNIL, WRITE delimits with **escape characters** print names containing lower-case, whitespace, macro, or escape characters, and print names beginning with decimal digits. If *PRINT-ESCAPE* is NIL, WRITE does not delimit print names by escape characters.

Delimiting the print name of such symbols by escape characters is essential for programs that generate source files, so the symbols can be correctly read by READ (see Section 5.8.1).

```
$ (SETQ *PRINT-ESCAPE* NIL)                    ;Turn off escape chars
NIL

$ '|I think, therefore I am.|                  ;Display w/o escape chars
I think, therefore I am.

$ (SETQ *PRINT-ESCAPE* T)                      ;Turn on escape chars
T

$ '|I think, therefore I am.|                  ;Display with escape chars
|I think, therefore I am.|
```

PRINT-BASE: 10 Control variable

If *PRINT-BASE* is an integer between 2 and 36 inclusive, WRITE, PRIN1, PRINT, and PRINC use that integer as the **output radix base** to represent numbers. Otherwise, these functions use radix base ten to represent numbers.

PRINT-BASE only affects the representation of numbers being sent to an output sink. It does *not* affect the binary representation used to store ratios internally.

The initial value of *PRINT-BASE* is 10. *READ-BASE* controls the input radix base used when reading numbers.

```
$ (SETQ TEN 10)                                    ;Initialize TEN to 10
10

$ (SETQ *PRINT-BASE* 16)                          ;Print numbers in hexadecimal
10

$ TEN                                                ;Ten in hexadecimal
0A

$ (SETQ *PRINT-BASE* 2)                           ;Print numbers in binary
10

$ TEN                                                ;Ten in binary
1010

$ (SETQ *PRINT-BASE* 10)                          ;Print numbers in decimal
10
```

PRINT-POINT: 7 Control variable

If *PRINT-POINT* is a nonnegative integer, WRITE, PRIN1, PRINT, and PRINC use **decimal notation** to represent ratios with up to *PRINT-POINT* digits following the decimal point. Otherwise, these functions use **rational notation** (i.e. the numerator, a /, and the denominator) to represent ratios exactly.

PRINT-POINT only affects the representation of ratios being sent to an output sink. It does *not* affect the precision used to store ratios internally.

The initial value of *PRINT-POINT* is 7. Numbers can be entered using either decimal or rational notation regardless of the value of *PRINT-POINT*.

```
$ (SETQ PI 3.1416)                          ;Initialize PI to 3.1416
3.1416
$ (SETQ *PRINT-POINT* NIL)                  ;Print using rational notation
NIL
$ PI                                        ;PI in rational notation
3927/1250
$ (SETQ *PRINT-POINT* 7)                    ;Print up to 7 digits
7
```

5.8.6 Console Screen

The functions in this subsection facilitate the display and formatting of text and graphics on the console screen. The methods used to accomplish these screen operations are dependent on the type of computer being used. Therefore when **muLISP** starts up, it tries to determine the **computer type** on which it is running by examining the computer's "fingerprint" stored in **read-only memory** (ROM). If successful, the second line of the header message (see Section 2.1) displays the computer type that was determined.

However, if the computer type cannot be determined, the prompt

```
1 = Other generic MS-DOS computer
2 = IBM PC compatible computer
3 = ANSI.SYS screen or VT-100 Terminal
4 = TI Professional Computer
5 = Zenith Z-100 Computer or VT-52 Terminal
6 = Hewlett-Packard HP-150 Computer
7 = Hewlett-Packard HP-110 Computer
8 = NEC Advanced PC or ADM-3A Terminal
9 = NEC PC-9801 Computer
A = Fujitsu Computer

Please enter your computer type number:
```

is displayed. If your computer type is not listed, you can try the various types to see if one of them makes the console screen functions work correctly on your computer. Choose type 1 only if all else fails, since the console screen functions and hence the **muLISP** Program Development System (see Chapter 7) will not work using this type.

If you become annoyed at having to enter your computer type each time **muLISP** starts, you can modify MULISP.COM so it automatically uses the correct computer type. Use the MS-DOS debugger DEBUG.COM to modify MULISP.COM as described in the introduction to Chapter 9.

A program can determine the computer type number stored at offset 855 in the **muLISP** base page (see Appendix H) decimal by making the call

```
(CSMEMORY 855)
```

MAKE-WINDOW [*row*, *column*, *rows*, *columns*] Function

The **current window** is a rectangular region of the console screen to which character and graphics output is confined. The **home position** is the top left corner of the current window. The **base row** and **base column** of the current window are respectively the row and column of the home position *relative* to the full console screen. Initially, the current window is the whole console screen.

Coordinates given to the cursor positioning, scrolling, and graphics functions are *relative* to the home position of the current window. Thus, the coordinates of the home position are row 0 and column 0.

The following discussion assumes a console screen of 25 rows by 80 columns. For other screen sizes simply replace these dimensions by ones appropriate for your console screen.

If available for your computer type (currently all types but generic MS-DOS), and if *row*, *column*, *rows*, and *columns* are within range, (MAKE-WINDOW *row column rows columns*) establishes a new current window, positions the cursor at the home position, and returns T. The base row of the new current window is *row*, the base column is *column*, the height is *rows*, and the width is *columns*.

row must be zero or a positive integer less than 25. *column* must be zero or a positive integer less than 80. Both *row* and *column* default to 0.

rows must be a positive integer less than or equal to 25 minus *row*. *rows* defaults to 25 minus *row*. *columns* must be a positive integer less than or equal to 80 minus *column*. *columns* defaults to 80 minus *column*.

(MAKE-WINDOW) returns a six element list consisting of the base row, the base column, the number of rows, and the number of columns in the current window, the height of a character in pixels, and the width of a character in pixels.

```
$ (MAKE-WINDOW 6 20 12 40)                ;Make a 12x40 window near middle of screen
T

$ (MAKE-WINDOW)                                        ;Current window coordinates
(6 20 12 40 1 1)                          (pixel height and width may vary)

$ (MAKE-WINDOW NIL NIL NIL NIL)                        ;Make a full-screen window
T
```

SET-CURSOR [*row, column*] Function
ROW [ ] Function
COLUMN [ ] Function
ROWS [ ] Function
COLUMNS [ ] Function

If available for your computer type (currently all types but generic MS-DOS), and if *row* and *column* are zero or positive integers less than the numbers of rows and columns respectively in the current window, (SET-CURSOR *row column*) repositions the cursor to the specified row and column in the current window and returns T. Otherwise, SET-CURSOR returns NIL.

(ROW) and (COLUMN) respectively return the row and column position of the cursor relative to the current window.

(ROWS) and (COLUMNS) respectively return the number of rows and columns in the current window.

```
(DEFUN SET-CURSOR (ROW COL)
  ((AND (<= 0 ROW 24) (<= 0 COL 79))
    (move cursor to ROWth row and COLth column in the current window)
    T ) )

(DEFUN ROW ()
  (cursor row in the current window) )

(DEFUN COLUMN ()
  (cursor column in the current window) )

$ (SET-CURSOR 0 0)                                          ;Home cursor
T

$ (ROW)                                                   ;Cursor on row 3
3

$ (COLUMN)                                              ;Cursor at left margin
0

$ (ROWS)                                                  ;25 rows in window
25

$ (COLUMNS)                                             ;80 columns in window
80
```

CLEAR-SCREEN [ ] Function
INSERT-LINES [*n*] Function
DELETE-LINES [*n*] Function

If available for your computer type (currently all types but generic MS-DOS), (CLEAR-SCREEN) clears the current window, moves the cursor to the home position, and returns T. Otherwise, CLEAR-SCREEN returns NIL.

If running on a PC compatible or PC-9801 compatible computer and *n* is a positive integer, (INSERT-LINES *n*) inserts *n* blank lines in the current window at the current cursor row and returns T. Otherwise, INSERT-LINES returns NIL. The lines are inserted by scrolling lines down from the current line toward the bottom of the screen.

If running on a PC compatible or PC-9801 compatible computer and *n* is a positive integer, (DELETE-LINES *n*) deletes *n* lines in the current window beginning with the current cursor row and returns T. Otherwise, DELETE-LINES returns NIL. The lines are deleted by scrolling lines up from below the cursor.

```
(DEFUN CLEAR-SCREEN ()
  (clear the current window of text)
  (SET-CURSOR 0 0) )

(DEFUN INSERT-LINES (N)
  ((AND (INTEGERP N) (>= N 0))
    (insert N blank lines at the current cursor row in the current window)
    T ) )

(DEFUN DELETE-LINES (N)
  ((AND (INTEGERP N) (>= N 0))
    (delete N lines beginning with the current cursor row in the current window)
    T ) )

$ (CLEAR-SCREEN)                                    ;Clear screen, home cursor
T

$ (SET-CURSOR 0 0)                                          ;Home cursor
T

$ (INSERT-LINES 5)                                        ;Insert 5 lines
T

$ (DELETE-LINES 5)                                        ;Delete 5 lines
T
```

VIDEO-MODE [*n*] Function

If running on a PC compatible computer and *n* is an integer between 0 and 255 inclusive, (VIDEO-MODE *n*) attempts to set the video mode to *n*, sets the active display page to the visual display page, makes the current window the full screen, establishes the normal, two-line cursor, and clears the screen. If the video mode is successfully set to *n*, VIDEO-MODE returns the previous video mode, otherwise it returns NIL.

(VIDEO-MODE) returns the current video mode.

There are numerous text and graphics **video modes** on PC compatible computers. Appendix A summarizes the resolution and the number of colors supported by each video mode. For more detailed information on video modes see the relevant hardware manuals.

Video modes 126 and 127 are for computers with **Hercules compatible graphics** adapter cards. To use these video modes, you must run the resident program HERCULES.COM *before* starting **muLISP**.

```
(DEFUN VIDEO-MODE (N)
  ((AND (INTEGERP N) (>= N 0))
    (attempt to set video mode to N)                    ;Set new video mode
    (DISPLAY-PAGE (DISPLAY-PAGE))          ;Make active page same as visual page
    (MAKE-WINDOW NIL)                                   ;Make full screen window
    (CURSOR-LINES NIL)                                ;Make normal two-line cursor
    (CLEAR-SCREEN)                                      ;Clear console screen
    (EQL (VIDEO-MODE) N) )                             ;Return T iff successful
  (return current video mode) )

$ (VIDEO-MODE)                                        ;Returns current video mode
7
```

FOREGROUND-COLOR [*n*] Function
BACKGROUND-COLOR [*n*] Function
PALETTE-COLOR [*n*] Function
BORDER-COLOR [*n*] Function

These color setting functions are only available if running on a PC compatible computer. Appendix B summarizes the color associated with each color number for the various PC video modes.

If *n* is an integer between 0 and 255 inclusive, (FOREGROUND-COLOR *n*) sets the foreground color used to display characters to *n* and returns the previous color. (FOREGROUND-COLOR) returns the current foreground color.

If *n* is an integer between 0 and 255 inclusive, (BACKGROUND-COLOR *n*) sets the background color used to display characters to *n* and returns the previous color. (BACKGROUND-COLOR) returns the current background color. In any text video mode (see Appendix A), a background color greater than 7 results in the blinking of characters.

If *n* is an integer between 0 and 255 inclusive, (PALETTE-COLOR *n*) attempts to set the palette color number to *n*. If successful, PALETTE-COLOR returns the previous palette color number, otherwise it returns NIL. (PALETTE-COLOR) returns the current palette color number. In some video modes, the palette color determines the set of available colors (see Appendix B).

If *n* is an integer between 0 and 255 inclusive, (BORDER-COLOR *n*) sets the border color to *n*. If successful, BORDER-COLOR returns the previous border color, otherwise it returns NIL. (BORDER-COLOR) returns the current border color. In text video modes, the border color determines the color of the border around the screen. In graphics video modes, the border color determines the color associated with color number 0 (i.e. the background color).

```
$ (PROGN (BACKGROUND-COLOR 7) (FOREGROUND-COLOR 0))          ;Set inverse video
7

$ (PALETTE-COLOR 1)                                         ;Use palette number 1
0

$ (BORDER-COLOR 7)                                           ;Use border color 7
0
```

DISPLAY-PAGE [*n*, *active-only*] Function

If running on a PC compatible computer and the graphics adapter card provides more than one video display page in the current video mode, DISPLAY-PAGE can be used to rapidly switch between display pages.

The page that is currently being displayed on the console screen is the **visual display page**. The page that is currently receiving output sent to the console is the **active display page**. For most applications the visual and active display pages are the same. However if output has been sent to an active page that is other than the visual page, switching the visual page to the active page "instantly" displays the output on the screen. As shown in the example below, this technique can be used to briefly display "subliminal" messages on the screen.

If *n* is an integer between 0 and 255 inclusive, (DISPLAY-PAGE *n*) attempts to set the active *and* visual display page to *n*. If successful, DISPLAY-PAGE returns the previous visual display page, otherwise it returns NIL.

If *n* is an integer between 0 and 255 inclusive and *active-only* is nonNIL, (DISPLAY-PAGE *n active-only*) attempts to set the active display page to *n*. If successful, DISPLAY-PAGE returns the previous active display page, otherwise it returns NIL.

(DISPLAY-PAGE) returns the current visual display page. (DISPLAY-PAGE NIL T) returns the current active display page.

```
(DEFUN DISPLAY-PAGE (N FLAG)
  ((AND (INTEGERP N) (>= N 0))
    ((NOT FLAG)
      (attempt to set active and visual display page to N)
      (EQL (DISPLAY-PAGE) N) )
    (attempt to set active display page to N)
    (EQL (DISPLAY-PAGE NIL T) N) )
  ((NOT FLAG)
    (return current visual display page) )
  (return current active display page) )

$ (DISPLAY-PAGE)                              ;Returns current visual display page
0

$ (DEFUN SUBLIMINAL-MESSAGE (MSG)
    (DISPLAY-PAGE 1 T)                                  ;Switch active page to 1
    (CLEAR-SCREEN)
    (SET-CURSOR 12 (FLOOR (- 80 (LENGTH MSG)) 2))
    (WRITE-STRING MSG)                                        ;Write to page 1
    (DISPLAY-PAGE 1)                                   ;Switch visual page to 1
    (TONE NIL 100)                                         ;Pause 1/10 second
    (DISPLAY-PAGE 0) )                                      ;Return to page 0
```

CURSOR-LINES [*start-line*, *end-line*] Function

The console **cursor** is a blinking block of light on the screen made up of one or more horizontal video lines one column wide. If running on a PC compatible computer, the starting and ending lines of the cursor can be adjusted under software control. If using a **monochrome display monitor**, an **enhanced graphics display monitor**, or a **multi-sync display monitor**, the cursor can consist of up to 14 lines. If using a **color graphics display monitor**, the cursor can consist of up to 8 lines. In either case, the cursor lines are numbered from the top starting with line 0.

If running on a PC compatible computer and if *start-line* and *end-line* are non-negative integers less than the maximum number of cursor lines for your monitor, (CURSOR-LINES *start-line end-line*) sets the cursor's start line and end line, and returns T.

(CURSOR-LINES 32 0) turns off the cursor display entirely. Otherwise, making *end-line* less than *start-line* may result in some rather bizarre cursor shapes depending on the video mode and graphics adapter hardware being used.

(CURSOR-LINES NIL) restores the cursor to the standard, two line cursor shape (lines 11 and 12 for monochrome and enhanced graphics monitors, 6 and 7 for color graphics monitors). When **muLISP** is terminated, the cursor shape is restored to the shape present at the time **muLISP** was started.

(CURSOR-LINES) returns a list of the cursor's current start line and end line. These examples assume a CGA, EGA, or VGA adapter card is being used in text mode (see Appendix A):

```
$ (CURSOR-LINES 0 13)                          ;Make cursor a full block
T

$ (CURSOR-LINES 32 0)                                ;Turn off cursor
T

$ (CURSOR-LINES NIL)                            ;Restore standard shape
T

$ (CURSOR-LINES)                              ;Return standard shape lines
(6 7)
```

READ-DOT [x, y] Function
PLOT-DOT [x, y, *color*] Function
PLOT-LINE [x_1, y_1, x_2, y_2, *color*] Function
PLOT-CIRCLE [x, y, r, *color*] Function
PLOT-COLOR: 15 Control variable

Graphics coordinates are always relative to the home position of the current window (see MAKE-WINDOW). Thus (0,0) is the coordinate pair of the home position. The maximum value of x is one less than the number of character columns in the current window times the width of a character in pixels. The maximum value of y is one less than the number of character rows in the current window times the height of a character in pixels. The width and height of a character in pixels for a given video mode can be determined by dividing the graphics resolution by the character resolution (see Appendix A) or by calling MAKE-WINDOW.

The number of available colors is determined by the current video mode (see Appendix A). Appendix B summarizes the color associated with each color number for the various PC video modes. If in a graphics video mode and *color* is greater than or equal to 128, dots are plotted in the color resulting from exclusive-or'ing their original color with *color*.

If (x,y) is a valid coordinate pair, (READ-DOT x y) returns the color of the dot at (x,y). If *color* is an integer between 0 and 255 inclusive, (PLOT-DOT x y *color*) plots a dot at (x,y) using color number *color*. If (x_1,y_1) and (x_2,y_2) are valid coordinates, (PLOT-LINE *x1 y1 x2 y2 color*) plots in *color* a line from (x_1,y_1) to (x_2,y_2). If r is a nonnegative integer, (PLOT-CIRCLE x y r *color*) plots in *color* a circle centered at (x,y) of radius r. Except for READ-DOT, these graphics functions are operative even in the text video modes, but at a very low resolution. *color* defaults to *PLOT-COLOR*.

```
$ (DEFUN PLOT-BOX (X1 Y1 X2 Y2 COLOR)
    (PLOT-LINE X1 Y1 X2 Y1 COLOR)
    (PLOT-LINE X2 Y1 X2 Y2 COLOR)
    (PLOT-LINE X2 Y2 X1 Y2 COLOR)
    (PLOT-LINE X1 Y2 X1 Y1 COLOR) )

$ (DEFUN FILL-BOX (X1 Y1 X2 Y2 COLOR)
    (LOOP
      (PLOT-LINE X1 Y1 X2 Y1 COLOR)
      ((= Y1 Y2))
      (INCQ Y1) ) )
```

5.8.7 Hardware Interface

The functions in this subsection facilitate communication between programs and facilities provided by the host computer's hardware and system software. These facilities are described in the technical reference manuals for the hardware and system software.

TIME [*reset*] Function
BENCHMARK [*form*] Library macro

(TIME) returns the time in 100ths of a second since an internal **real time clock** was last reset. If *reset* is nonNIL, (TIME *reset*) resets the clock and returns NIL. Resetting the internal real time clock does *not* affect the real time clock used by the operating system.

(BENCHMARK *form*) returns the result of evaluating *form* after displaying the time required for the evaluation. This macro is useful to **benchmark** programs so you can determine the most efficient algorithm to accomplish a given task. BENCHMARK is defined in COMMON.LSP.

```
(DEFUN TIME (RESET)
  ((NULL RESET)
    (return the time in 100ths of a second since the clock was last reset) )
  (reset the clock)
  NIL )

(DEFMACRO BENCHMARK (FORM)
  (LIST 'PROGN '(TIME T) (LIST 'UNWIND-PROTECT
                               FORM
                               '(PRIN1 (/ (TIME) 100))
                               '(WRITE-LINE " seconds"))) )

$ (BENCHMARK (RECLAIM))                          ;Benchmark the garbage collector
0.5 seconds                                                ;(results will vary)
330151
```

GET-DECODED-TIME [ ] Function
GET-UNIVERSAL-TIME [ ] Library function
TIME-ZONE: 0 Library variable
ENCODE-UNIVERSAL-TIME [*second, minute, hour, day, month, year, zone*] Library function

(GET-DECODED-TIME) returns the current **calender time** (decoded time) accurate to the nearest second. Calender time is represented as a list of six integers signifying the following: year (A.D.), month (1-12), day (1-31), hour (0-23), minute (0-59), and second (0-59).

(GET-UNIVERSAL-TIME) returns the current **universal time** as an integer accurate to the nearest second. Universal time is the number of seconds since midnight, January 1, 1900 GMT. GET-UNIVERSAL-TIME uses *TIME-ZONE* to determine the current time zone.

(ENCODE-UNIVERSAL-TIME *second minute hour day month year zone*) converts the specified calender time to the equivalent universal time. If *zone* is not given, ENCODE-UNIVERSAL-TIME uses *TIME-ZONE* to determine the current time zone.

TIME-ZONE is a variable that should be set to the number of hours that your time is behind **Greenwich Mean Time** (GMT). Use a negative number of hours if your time is ahead of GMT. The initial value of *TIME-ZONE* is 0.

TIME-ZONE and the universal time functions are defined in COMMON.LSP.

```
(DEFUN GET-DECODED-TIME ()
  (a list of the current year, month, day, hour, minute, and second) )

(DEFUN GET-UNIVERSAL-TIME ()
  (APPLY 'ENCODE-UNIVERSAL-TIME (NREVERSE (GET-DECODED-TIME))) )

$ (GET-DECODED-TIME)                    ;Decoded calendar time of the 1989 New Year
(1989 1 1 0 0 0)

$ (GET-UNIVERSAL-TIME)                          ;Universal time of the 1989 New Year
2808604800

$ (ENCODE-UNIVERSAL-TIME 0 0 0 1 1 2000)          ;Time of the 2000 New Year in GMT
3155673600

$ (ENCODE-UNIVERSAL-TIME 0 0 0 1 1 2000 10)       ;Time of the 2000 New Year in Hawaii
3155709600
```

TONE [*frequency, milliseconds*] Function
SLEEP [*seconds*] Library function

If running on a PC compatible computer, and *frequency* is an integer between 37 and 32767 inclusive, and *n* is a nonnegative integer, (TONE *frequency n*) generates a tone of *frequency* Hertz (cycles per second) for *n* milliseconds (thousandths of a second). If running on other than a PC compatible or if *frequency* is not an integer in the correct range, TONE causes program execution to cease for *n* milliseconds without producing any tone. If *n* is a nonnegative integer, TONE returns T, otherwise it returns NIL.

The following table gives the frequency in Hertz corresponding to notes for octaves on either side of middle C (262 Hertz):

Note	Frequency			
C	131	262	523	1046
D	147	294	587	1175
E	165	330	659	1318
F	175	349	698	1397
G	196	392	784	1568
A	220	440	880	1760
B	247	494	988	1975

When **muLISP** starts, it uses the host computer's system clock to calibrate TONE so it will generate a tone or pause for the correct duration of time. Since this calibration can only be an approximation, the duration of a call on TONE may be off by some constant factor.

If *n* is a nonnegative number, (SLEEP *n*) causes program execution to cease temporarily for approximately *n* seconds. After the pause, SLEEP returns NIL. SLEEP is defined in COMMON.LSP.

```
(DEFUN TONE (FREQ N)
  ((AND (INTEGERP N) (>= N 0))
    ((AND (INTEGERP FREQ) (<= 37 FREQ 32767))
      (generate FREQ Hertz tone for N milliseconds) )
    (pause for N milliseconds) ) )

(DEFUN SLEEP (N)
  (TONE NIL (TRUNCATE N 1/1000))
  NIL )

$ (PROGN                                          ;Generate a familiar movie theme
    (TONE 784 600) (TONE 880 600) (TONE 698 600)
    (TONE 349 600) (TONE 523 900) (TONE NIL 600))
```

PORTIO [*port, value, flag*] Function

If *port* is an integer between 0 and 65535 inclusive, (PORTIO *port*) inputs and returns a byte value from *port*. If *value* is an integer between 0 and 255 inclusive, (PORTIO *port value*) outputs *value* to *port* and returns NIL.

If *flag* is nonNIL, (PORTIO *port* NIL *flag*) inputs and returns a word value from *port*. If *value* is an integer between 0 and 65535 inclusive, (PORTIO *port value*) outputs *value* to *port* and returns NIL.

The ability to send and receive bytes and words from ports is useful for programs that need to communicate with external devices such as sensors and robots.

```
(DEFUN PORTIO (PORT VALUE FLAG)
  ((AND (INTEGERP PORT) (<= 0 PORT 65535))
    ((NULL FLAG)
      ((AND (INTEGERP VALUE) (<= 0 VALUE 255))
        (output byte VALUE to PORT)
        NIL )
      (input a byte value from PORT) )
    ((AND (INTEGERP VALUE) (<= 0 VALUE 65535))
      (output word VALUE to PORT)
      NIL )
    (input a word value from PORT) ) )

$ (PORTIO 70)                                    ;Input byte from port 70
208                                                 ;(results will vary)

$ (PORTIO 70 NIL T)                              ;Input word from port 70
36610                                               ;(results will vary)

$ (PORTIO 70 10)                                  ;Output byte to port 70
NIL

$ (PORTIO 70 10 T)                               ;Output word to port 70
NIL
```

MEMORY [*address*, *value*, *flag*] Function

If *address* is an integer between 0 and 1,048,575 inclusive (the 20 bit address space of the 8086 microprocessor), (MEMORY *address*) returns the contents of the byte at *address* as an integer between 0 and 255 inclusive. If *value* is an integer between 0 and 255 inclusive, (MEMORY *address value*) stores *value* in the byte at *address*, and returns the original contents of the byte.

If *flag* is nonNIL, (MEMORY *address* NIL *flag*) returns the contents of the bytes at *address* and *address*+1 as an integer between 0 and 65535 inclusive. If *value* is an integer between 0 and 65535 inclusive, (MEMORY *address value flag*) stores *value* in the bytes at *address* and *address*+1, and returns the original contents of the bytes.

MEMORY *should be used with care since there is no protection against inadvertently destroying* **muLISP** *or the host operating system.*

```
(DEFUN MEMORY (ADDR VALUE FLAG)
  ((AND (INTEGERP ADDR) (<= 0 ADDR 1048575))
    ((NULL FLAG)
      ((AND (INTEGERP VALUE) (<= 0 VALUE 255))
        (replace the byte at ADDR with VALUE)
        (return the original byte at ADDR) )
      (return the byte at ADDR) )
    ((AND (INTEGERP VALUE) (<= 0 VALUE 65535))
      (replace the word at ADDR with VALUE)
      (return the original word at ADDR) )
    (return the word at ADDR) ) )

$ (SETQ *PRINT-BASE* 16 *READ-BASE* 16)                    ;Set hexadecimal I/O
10

$ (MEMORY 0A)                                              ;Get byte at location 0AH
0

$ (MEMORY 0A NIL T)                                        ;Get word at location 0AH
0F000

$ (MEMORY 0A 5)                                          ;Store a byte at location 0AH
0

$ (MEMORY 0A NIL T)                                        ;Get word at location 0AH
0F005

$ (MEMORY 0A 0)                                          ;Restore original byte at 0AH
5

$ (SETQ *PRINT-BASE* 0A *READ-BASE* 0A)                          ;Set decimal I/O
10
```

CSMEMORY [*offset, value, flag*] Function
DSMEMORY [*offset, value, flag*] Function

These functions are similar to MEMORY, except their first argument is a 16 bit offset into a segment rather than a 20 bit physical memory address. The CSMEMORY offset is relative to the code segment. The DSMEMORY offset is relative to the data segment.

If *offset* is an integer between 0 and 65535 inclusive, (CSMEMORY *offset*) and (DSMEMORY *offset*) return the contents of the byte at *offset* as an integer between 0 and 255 inclusive. If *value* is an integer between 0 and 255 inclusive, (CSMEMORY *offset value*) and (DSMEMORY *offset value*) store *value* in the byte at *offset*, and return the original contents of the byte.

If *flag* is nonNIL, (CSMEMORY *offset* NIL *flag*) and (DSMEMORY *offset* NIL *flag*) return the contents of the bytes at *offset* and *offset*+1 as an integer between 0 and 65535 inclusive. If *value* is an integer between 0 and 65535 inclusive, (CSMEMORY *offset value flag*) and (DSMEMORY *offset value flag*) store *value* in the bytes at *offset* and *offset*+1, and return the original contents of the bytes.

(CSMEMORY) and (DSMEMORY) return the contents of the current code segment and data segment registers respectively.

CSMEMORY *and* DSMEMORY *should be used with care since there is no protection against inadvertently destroying* **muLISP**.

```
(DEFUN CSMEMORY (OFFSET VALUE FLAG)
  ((AND (INTEGERP OFFSET) (<= 0 OFFSET 65535))
    ((NULL FLAG)
      ((AND (INTEGERP VALUE) (<= 0 VALUE 255))
        (store VALUE in the byte at OFFSET)
        (return the original contents of the byte at OFFSET) )
      (return the contents of the byte at OFFSET) )
    ((AND (INTEGERP VALUE) (<= 0 VALUE 65535))
      (store VALUE in the bytes at OFFSET and OFFSET+1)
      (return the original contents of the bytes) )
    (return the contents of the bytes at OFFSET and OFFSET+1) )
  (return the current contents of the CS register) )
```

CSMEMORY is useful for accessing the constants and tables stored in the **muLISP** base page (see Appendix H). The multiple escape character that **muLISP** uses to delimit print name strings containing special characters is stored in the base page at offset 849. This example shows how to change the multiple escape character.

```
$ (ASCII (CSMEMORY 849))
\|                                          ;Default multiple escape char

$ "LISP is fun."
|LISP is fun.|                              ;String delimited by | char

$ (CSMEMORY 849 (ASCII \"))                 ;Change multiple escape char
124                                         ;Return original char

$ "LISP is fun."
"LISP is fun."                              ;String delimited by " char

$ (CSMEMORY 849 124)
34                                  ;Restore original multiple escape char

$ (CSMEMORY)                                ;The muLISP code segment register
8266
```

DSMEMORY is useful for accessing the variables stored in the **muLISP** global storage area (see Appendix I). **Warning**: The value of variables in the global storage area should *not* be changed. The beginning and ending addresses of the pseudo-code is stored as words in the Global Storage Area at offsets 34 and 36 respectively. This example shows how to determine the current size of the pseudo-code in bytes (naturally the results will vary).

```
$ (DSMEMORY 34 NIL T)                       ;Base of pseudo-code
5346

$ (DSMEMORY 36 NIL T)                       ;End of pseudo-code
24534

$ (- * **)                                  ;Bytes of pseudo-code
19188

$ (DSMEMORY)                                ;The muLISP data segment register
20451
```

LOCATION [*object*] Function

The pointer cells making up an object are stored in separate data segments at the same offset address. (LOCATION *object*) returns the offset of *object*'s cells from the base of the data segments. Since all cells require exactly two bytes, LOCATION always returns an even integer.

Because address typing is used to determine the type of an object, the location of a symbol will be less than the location of a number, which will be less than the location of a cons. The automatic compacting garbage collector and memory reallocator can change the location of an object at any time. LOCATION is primarily intended for compilers that generate machine code which must refer to the location of symbols that have a fixed offset in memory.

```
(DEFUN LOCATION (OBJECT)
  (the offset address of OBJECT) )

$ (LOCATION NIL)                                    ;Location of type symbol NIL
256                                                 ;(Note: NIL is fixed at 100H)

$ (LOCATION 2)                                      ;Location of a number
11622

$ (LOCATION '(A . B))                               ;Location of a cons
37838
```

REGISTER [*regnum*, *low*, *high*] Function
INTERRUPT [*n*] Function

These functions enable programs to communicate with external **machine language** routines via the 256 **interrupt vectors** provided by the Intel 8086 or compatible microprocessors. The computer's technical reference and operating system manuals describe the services provided by the interrupt vectors.

There are 10 pseudo-registers numbered 0 through 9. They correspond to the 8086 registers as follows:

Pseudo:	0	1	2	3	4	5	6	7	8	9
8086:	AX	BX	CX	DX	SI	DI	BP	DS	ES	FLAG

If *regnum* is an integer between 0 and 9 inclusive, (REGISTER *regnum*) returns the contents of pseudo-register *regnum*. If *low* is an integer between 0 and 65535 inclusive, (REGISTER *regnum low*) stores *low* in pseudo-register *regnum*, and returns NIL. If *high* is an integer between 0 and 255 inclusive, (REGISTER *regnum low high*) stores *low* + 256 *high* in pseudo-register *regnum*, and returns NIL. The initial value for all the pseudo-registers is 0.

If *n* is an integer between 0 and 255 inclusive, (INTERRUPT *n*) loads the microprocessor's registers from the pseudo-registers, executes interrupt number *n*, stores the register values returned by the interrupt in the pseudo-registers, and returns NIL.

For example, Interrupt 33 (i.e. 21H) is used to make MS-DOS **system calls**. If register AH is 42 (i.e. 2AH), Interrupt 33 will return the year in CX (1980-2099), the month in DH (1-12), and the day in DL (1-31). Using this system call, GET-DATE defined below returns the current year, month, and day as a list of three integers (see also GET-DECODED-TIME).

```
(DEFUN REGISTER (N M)
  ((AND (INTEGERP N) (<= 0 N 9))
    ((AND (INTEGERP M) (<= 0 M 65535))
      (store M in register N)
      NIL )
    (the integer value of register N) ) )

$ (DEFUN GET-DATE ()
    (REGISTER 0 (* 42 256))                    ;Set AH to 42
    (INTERRUPT 33)                            ;Make MS-DOS call
    (LIST (REGISTER 2)                         ;CX has year
          (SHIFT (REGISTER 3) -8)              ;DH has month
          (LOGAND (REGISTER 3) 255)) )         ;DL has day

$ (GET-DATE)                                  ;The 1989 New Year
(1989 1 1)
```

BINARY-LOAD [*filename, offset*] Function

If *offset* is an integer between 0 and 65535 inclusive and the file on the drive and directory specified by *filename* exists, (BINARY-LOAD *filename offset*) loads *filename* into the **muLISP** code segment at *offset* and returns the number of bytes loaded. If no file name extension is specified, the extension BIN is assumed. If *offset* is out of range or no file specified by *filename* exists, BINARY-LOAD returns NIL.

BINARY-LOAD is normally used in conjunction with the function ALLOCATE (see Section 5.7.6) to load externally assembled or compiled machine language routines into memory in the **muLISP** code segment. Section 9.4 gives an example of its use.

```
(DEFUN BINARY-LOAD (FILE-NAME OFFSET)
  ((AND (INTEGERP OFFSET) (<= 0 OFFSET 65535))
    ((PROBE-FILE FILE-NAME)
      (load a binary image of FILE-NAME into the code segment beginning at OFFSET)
      (return the number of bytes loaded) ) ) )
```

SNAPSHOT [*address*, *atom*] Function

If *address* is an integer between 0 and 1,048,575 inclusive (the 20 bit address space of the 8086 microprocessor) and *atom* is an integer between 0 and 65535 inclusive, (SNAPSHOT *address atom*) returns the symbol whose print name consists of the *atom* bytes of memory beginning at *address*. If *atom* is a symbol, (SNAPSHOT *address atom*) copies the bytes in *atom*'s print name to memory beginning at *address*, and returns T.

SNAPSHOT is especially useful for saving and later restoring memory-mapped video screen memory. The **monochrome display adapter** (MDA) card stores screen memory beginning at 0B0000H. The PC **color graphics adapter** (CGA) card stores screen memory beginning at 0B8000H. See your computer's reference manual to learn how and where it stores screen memory.

```
(DEFUN SNAPSHOT (ADDR ATOM)
  ((AND (INTEGERP ADDR) (<= 0 ADDR 1048575))
    ((SYMBOLP ATOM)
      (store the bytes in ATOM's print name in memory beginning at ADDR)
      T )
    ((AND (INTEGERP ATOM) (<= 0 ATOM 65535))
      (return the symbol whose print name consists of ATOM bytes of memory
          beginning at ADDR)
       ) ) )

$ (SETQ *READ-BASE* 16)                                    ;Input hexadecimal
16

$ (SETQ PIX (SNAPSHOT 0B0000 200) NIL)                        ;Save picture
NIL

$ (SNAPSHOT 0B1000 PIX)                                    ;Redisplay picture
T

$ (SETQ *READ-BASE* 0A)                                       ;Input decimal
10
```

5.9 File System

The file system primitives enable programs to open and close files for input and/or output, select a file for standard input and/or output, adjust the read and/or write position of open files, and perform standard file and directory operations. The only limit on the number of files that can be open at any one time is that imposed by the host operating system (see the MS-DOS manual).

Most of the file system functions are given one or more file pathnames as arguments and/or return one or more pathnames as values. A full **pathname** consists of a drive letter, a directory name, a file name, and a file extension. Standard MS-DOS syntax is used for pathnames. For example,

```
C:\MULISP\ANIMAL.LSP
```

specifies the file ANIMAL.LSP on drive C in the directory MULISP.

The **backslash character** used as a separator in pathnames is also the multiple escape character used by the input functions (see Section 5.8.1). Thus, two backslashes must be entered for each backslash in the file's name. For example,

```
(LOAD C:\\MULISP\\ANIMAL.LSP)
```

is the command to load the above file using its full pathname.

If no drive is included in a pathname given to a file system function, the **default drive** is assumed. If no directory is included in a pathname, the **current directory** is assumed. For example,

```
(LOAD ANIMAL.LSP)
```

is the command to load ANIMAL.LSP from the current directory of the default drive.

Certain file names are reserved by MS-DOS as system **device names**. These include CON, AUX, COM1, COM2, LPT1, LPT2, LPT3, PRN, and NUL (see the MS-DOS manual). If a file system function is given one of these names as a pathname, the corresponding system device will be used instead of a file.

5.9.1 Input Files

The input file primitives enable programs to open and close files for input, select the **standard input file** and the **standard input source**, and adjust the read position of open input files.

If no input source is given in a call to an input function (see Section 5.8.1), the standard input source is used. The standard input source can be either the console or the standard input file. Although several files can be open for input, only *one* file at a time can be the standard input file.

OPEN-INPUT-FILE [*filename*] Function

If the file on the drive and directory specified by *filename* exists, (OPEN-INPUT-FILE *filename*) opens *filename* for input and makes it the standard input file, initializes its read position to 0, sets the control variable *INPUT-FILE* to its pathname (thus making *filename* the standard input source as well as the standard input file), and returns its pathname. If no file specified by *filename* exists, OPEN-INPUT-FILE returns NIL.

The example below opens the file ANIMAL.LSP for input and makes it the standard input file. Making ANIMAL.LSP the standard input file does *not* automatically load the file. Instead use the function LOAD to read in and evaluate the expressions in a source file.

```
(DEFUN OPEN-INPUT-FILE (FILE-NAME)
  ((PROBE-FILE FILE-NAME)
    (open FILE-NAME for input)
    (INPUT-FILE FILE-NAME)
    (FILE-READ-POSITION 0)
    (SETQ *INPUT-FILE* (INPUT-FILE)) ) )

$ (OPEN-INPUT-FILE 'ANIMAL.LSP)                          ;Open ANIMAL.LSP
C:ANIMAL.LSP

$ *INPUT-FILE*                                  ;*INPUT-FILE* is now nonNIL
C:ANIMAL.LSP

$ (READ-LINE)                                     ;Read first line of file
|;File: ANIMAL.LSP  (C)          12/19/88          Soft Warehouse, Inc.|
```

CLOSE-INPUT-FILE [*filename*] Function

If *filename* is an open input file, (CLOSE-INPUT-FILE *filename*) closes *filename* for input and returns T. If *filename* is also the standard input file, CLOSE-INPUT-FILE sets the control variable *INPUT-FILE* to NIL. If *filename* is not an open input file, (CLOSE-INPUT-FILE *filename*) returns NIL.

If there is a standard input file, (CLOSE-INPUT-FILE) closes the file for input, sets the control variable *INPUT-FILE* to NIL, and returns T. If there is no standard input file, (CLOSE-INPUT-FILE) returns NIL.

```
(DEFUN CLOSE-INPUT-FILE (FILE-NAME)
  ((NULL FILE-NAME)
    ((INPUT-FILE)
      (CLOSE-INPUT-FILE (INPUT-FILE)) ) )
  ((INPUT-FILE-P FILE-NAME)
    (IF (EQ FILE-NAME (INPUT-FILE))
        (SETQ *INPUT-FILE*) )
    (close FILE-NAME for input)
    T ) )

$ (CLOSE-INPUT-FILE)                          ;Close the standard input file
T

$ *INPUT-FILE*                                ;*INPUT-FILE* is now NIL
NIL
```

INPUT-FILE [*filename*] Function
INPUT-FILE-P [*filename*] Function
INPUT-FILES [ ] Function

If *filename* is an open input file, (INPUT-FILE *filename*) makes *filename* the standard input file and returns its pathname. If *filename* is not an open input file, (INPUT-FILE *filename*) returns NIL.

If there is a standard input file, (INPUT-FILE) returns the pathname of the file. If there is no standard input file, (INPUT-FILE) returns NIL.

If *filename* is an open input file, (INPUT-FILE-P *filename*) returns T, otherwise it returns NIL.

(INPUT-FILES) returns a list of the pathnames of the currently open input files.

```
(DEFUN INPUT-FILE (FILE-NAME)
  ((NULL FILE-NAME)
    ((if there is a standard input file)
      (return the pathname of the file) ) )
  ((INPUT-FILE-P FILE-NAME)
    (make FILE-NAME the standard input file)
    (return the pathname of FILE-NAME) ) )

(DEFUN INPUT-FILE-P (FILE-NAME)
  (return T if FILE-NAME is an open input file, otherwise NIL) )

(DEFUN INPUT-FILES ()
  (return a list of the open input files) )

$ (OPEN-INPUT-FILE 'ANIMAL.LSP)                          ;Open ANIMAL.LSP
C:ANIMAL.LSP

$ (OPEN-INPUT-FILE 'HANOI.LSP)                           ;Open HANOI.LSP
C:HANOI.LSP

$ (INPUT-FILE)                          ;HANOI.LSP is the standard input file
C:HANOI.LSP

$ (INPUT-FILE 'ANIMAL.LSP)             ;Make ANIMAL.LSP the standard input file
C:ANIMAL.LSP

$ (INPUT-FILE)                           ;ANIMAL.LSP is the standard input file
C:ANIMAL.LSP

$ (INPUT-FILE-P 'HANOI.LSP)                 ;HANOI.LSP is still open for input
T

$ (INPUT-FILES)                                    ;Currently open input files
(C:HANOI.LSP C:ANIMAL.LSP)
```

INPUT-FILE: NIL Control variable

If there is a standard input file and *INPUT-FILE* is nonNIL, standard input comes from the file; otherwise standard input comes from the console. If there is no standard input file, the console is the standard input source regardless of the value of *INPUT-FILE*.

Normally the functions OPEN-INPUT-FILE and INPUT-FILE provide sufficient control over the source of character input to programs. However, after reading part of a file, it may be necessary to temporarily receive input from the console before returning to the file. The control variable *INPUT-FILE* provides this capability.

After using OPEN-INPUT-FILE to open a file and make it the standard input file, the console can be made the standard input source without changing the standard input file by setting *INPUT-FILE* to NIL. A subsequent nonNIL assignment to *INPUT-FILE* will again make the standard input file the standard input source, and reading of the file will resume from the point at which it had been suspended.

An alternative method of temporarily receiving input from the console is to specify the console in calls on the input functions (see Section 5.8.1).

FILE-READ-POSITION [*n*, *filename*] Function

If *filename* is an open input file and *n* is zero or a positive integer, (FILE-READ-POSITION *n filename*) sets the read position of *filename* to *n* and returns T. If *n* is neither zero nor a positive integer, FILE-READ-POSITION returns the current read position of *filename*. If *filename* is not an open input file, FILE-READ-POSITION returns NIL. If no *filename* argument is given, the standard input file is assumed.

A file's **read position** determines the point in the file at which the next character will be read. When a file is opened for input, the file's read position is set to 0 (the first byte of the file). After a character is read from the file, the file's read position is automatically incremented. Thus FILE-READ-POSITION needs to be used only when a file must be accessed randomly.

A good example of the use of FILE-READ-POSITION is in the **muLISP Tutorial System** source file LESSONS.LSP. The system reads and displays the script of a lesson using the function READ-LESSON. Since the system provides an option for viewing the previous screen, READ-LESSON pushes the read position marking the beginning of each screen onto the list *POSN-LIST*. When the student wants to see the previous screen, the call

(FILE-READ-POSITION (CAR *POSN-LIST*))

is used to move the read position back to the beginning of the screen so it can be read and displayed again.

```
(DEFUN FILE-READ-POSITION (N FILE-NAME)
  ((NULL FILE-NAME)
    ((INPUT-FILE)
      (FILE-READ-POSITION N (INPUT-FILE)) ) )
  ((INPUT-FILE-P FILE-NAME)
    ((AND (INTEGERP N) (>= N 0))
      (set the read position of FILE-NAME to N)
      T )
    (FILE-NAME current read position) ) )

$ (OPEN-INPUT-FILE 'ANIMAL.LSP)                          ;Open ANIMAL.LSP
C:ANIMAL.LSP

$ (FILE-READ-POSITION)                                   ;Initial read position
0

$ (READ-LINE)                                            ;Read first line of file
|;File: ANIMAL.LSP  (C)          12/19/88                Soft Warehouse, Inc.|

$ (FILE-READ-POSITION)                                   ;Current read position
55
```

5.9.2 Output Files

The output file primitives enable programs to open and close files or output devices for output, select the **standard output file** and the **standard output sink**, and adjust the write position and line length of open output files.

If no output sink is given in a call to an output function (see Section 5.8.4), the standard output sink is used. The standard output sink can be either the console or the standard output file. Although several files can be open for output, only *one* file at a time can be the standard output file.

OPEN-OUTPUT-FILE [*filename, overwrite*] Function

In the event *filename* is already an open output file, (OPEN-OUTPUT-FILE *filename overwrite*) first closes *filename* for output before proceeding with the following operations:

If the file on the drive and directory specified by *filename* exists and *overwrite* is nonNIL, (OPEN-OUTPUT-FILE *filename overwrite*) opens *filename* for output and makes it the standard output file, initializes its write position to 0, sets the control variable *OUTPUT-FILE* to its pathname (thus making *filename* the standard output sink as well as the standard output file), and returns its pathname.

If the file specified by *filename* does not exist and *overwrite* is nonNIL, (OPEN-OUTPUT-FILE *filename overwrite*) first creates a new directory entry for *filename* before proceeding with the above. The overwrite flag makes it possible to simply reopen and modify existing files rather than having to write an entirely new file.

(OPEN-OUTPUT-FILE *filename*) opens a temporary file for output, leaving *filename* untouched until the temporary file is successfully closed. Specifically, OPEN-OUTPUT-FILE deletes the BAK file for *filename* if it should exist, creates a new directory entry for the temporary file, opens the temporary file for output and makes it the standard output file, initializes its write position to 0, sets the control variable *OUTPUT-FILE* to *filename* (thus making the temporary file the standard output sink as well as the standard output file), and returns *filename*. Note that the original file is not renamed as a BAK file *until* the temporary file is successfully closed (see CLOSE-OUTPUT-FILE). This automatic single level **file backup** feature prevents losing the original file should a disk error occur while writing to the new file.

If OPEN-OUTPUT-FILE attempts to create a new directory entry and there is no MS-DOS directory entry available, OPEN-OUTPUT-FILE returns NIL.

To direct output to a printer, call OPEN-OUTPUT-FILE using the MS-DOS device name for the printer in place of *filename*. PRN is the device name for parallel printer #1, LPT*n* is the name for parallel printer #*n*, and COM*n* is the name for serial printer #*n*. For example, (OPEN-OUTPUT-FILE 'LPT1) makes parallel printer #1 the standard output sink. (Note: To make a "hard copy" print-out of a **muLISP** session, use the control variable *PRINTER-ECHO* (see Section 5.8.5) instead of using OPEN-OUTPUT-FILE.)

```
(DEFUN OPEN-OUTPUT-FILE (FILE-NAME OVERWRITE)
  (CLOSE-OUTPUT-FILE FILE-NAME)
  ( ((NULL OVERWRITE)
     (delete BAK file for FILE-NAME if it exists)
     (create a temporary file for FILE-NAME) )
    ((PROBE-FILE FILE-NAME))
    (create a new file for FILE-NAME) )
  (open FILE-NAME for output)
  (OUTPUT-FILE FILE-NAME)
  (FILE-WRITE-POSITION 0)
  (SETQ *OUTPUT-FILE* (OUTPUT-FILE)) )

$ (OPEN-OUTPUT-FILE TEST.LSP)                    ;Make TEST.LSP the standard output file
C:TEST.LSP

$ *OUTPUT-FILE*                                          ;*OUTPUT-FILE* is now nonNIL
C:TEST.LSP

$ (FILE-WRITE-POSITION)                                  ;Initial write position
0

$ (WRITE-LINE '|This is a test.|)
|This is a test.|

$ (FILE-WRITE-POSITION)                                  ;Current write position
16
```

CLOSE-OUTPUT-FILE [*filename*] Function

If *overwrite* was nonNIL when *filename* was opened by OPEN-OUTPUT-FILE, (CLOSE-OUTPUT-FILE *filename*) closes *filename* for output, sets the length of the file to the current write position of *filename*, and returns T. If *filename* is also the standard output file, CLOSE-OUTPUT-FILE sets the control variable *OUTPUT-FILE* to NIL.

If *overwrite* was NIL when *filename* was opened by OPEN-OUTPUT-FILE, (CLOSE-OUTPUT-FILE *filename*) closes the temporary file created for *filename*, sets the length of the temporary file to the current write position of *filename*, renames the existing *filename* to be a BAK file, renames the temporary file to *filename*, and returns T. If *filename* is also the standard output file, CLOSE-OUTPUT-FILE sets the control variable *OUTPUT-FILE* to NIL.

If *filename* is not an open output file, (CLOSE-OUTPUT-FILE *filename*) returns NIL.

If there is a standard output file, (CLOSE-OUTPUT-FILE) closes the file as described above, and returns T. If there is no standard output file, (CLOSE-OUTPUT-FILE) returns NIL.

```
(DEFUN CLOSE-OUTPUT-FILE (FILE-NAME)
  ((NULL FILE-NAME)
    ((OUTPUT-FILE)
      (CLOSE-OUTPUT-FILE (OUTPUT-FILE)) ) )
  ((OUTPUT-FILE-P FILE-NAME)
    (IF (EQ FILE-NAME (OUTPUT-FILE))
        (SETQ *OUTPUT-FILE*) )
    (close FILE-NAME for output)
    (set the file length to write position)
    ((if FILE-NAME is an overwrite output file))
    (rename FILE-NAME to a BAK file)
    (rename the temporary file to FILE-NAME)
    T ) )

$ (CLOSE-OUTPUT-FILE)                          ;Close the standard output file
T

$ *OUTPUT-FILE*                                  ;*OUTPUT-FILE* is now NIL
NIL
```

OUTPUT-FILE [*filename*] Function
OUTPUT-FILE-P [*filename*] Function
OUTPUT-FILES [ ] Function

If *filename* is an open output file, (OUTPUT-FILE *filename*) makes *filename* the standard output file and returns its pathname. If *filename* is not an open output file, (OUTPUT-FILE *filename*) returns NIL.

If there is a standard output file, (OUTPUT-FILE) returns the pathname of the file. If there is no standard output file, (OUTPUT-FILE) returns NIL.

If *filename* is an open output file, (OUTPUT-FILE-P *filename*) returns T, otherwise it returns NIL.

(OUTPUT-FILES) returns a list of the pathnames of the currently open output files.

```
(DEFUN OUTPUT-FILE (FILE-NAME)
  ((NULL FILE-NAME)
    ((if there is a standard output file)
      (return the pathname of the file) ) )
  ((OUTPUT-FILE-P FILE-NAME)
    (make FILE-NAME the standard output file)
    (return the pathname of FILE-NAME) ) )

(DEFUN OUTPUT-FILE-P (FILE-NAME)
  (return T if FILE-NAME is an open output file, otherwise NIL) )

(DEFUN OUTPUT-FILES ()
  (return a list of the open output files) )

$ (OPEN-OUTPUT-FILE 'TEST1.LSP)                          ;Open TEST1.LSP
C:TEST1.LSP

$ (OPEN-OUTPUT-FILE 'TEST2.LSP)                          ;Open TEST2.LSP
C:TEST2.LSP

$ (OUTPUT-FILE)                          ;TEST2.LSP is the standard output file
C:TEST2.LSP

$ (OUTPUT-FILE 'TEST1.LSP)              ;Make TEST1.LSP the standard output file
C:TEST1.LSP

$ (OUTPUT-FILE)                          ;TEST1.LSP is the standard output file
C:TEST1.LSP

$ (OUTPUT-FILE-P 'TEST2.LSP)                ;TEST2.LSP is still open for output
T

$ (OUTPUT-FILES)                                    ;Currently open output files
(C:TEST2.LSP C:TEST1.LSP)
```

OUTPUT-FILE: NIL Control variable

If there is a standard output file and *OUTPUT-FILE* is nonNIL, standard output goes to the file; otherwise standard output goes to the console. If there is no standard output file, the console is the standard output sink regardless of the value of *OUTPUT-FILE*.

Normally the functions OPEN-OUTPUT-FILE and OUTPUT-FILE provide sufficient control over the destination of character output from programs. However, after writing part of a file, it may be necessary to temporarily send output to the console before returning to the file. The control variable *OUTPUT-FILE* provides this capability.

After using OPEN-OUTPUT-FILE to open a file and make it the standard output file, the console can be made the standard output sink without changing the standard output file by setting *OUTPUT-FILE* to NIL. A subsequent nonNIL assignment to *OUTPUT-FILE* will again make the standard output file the standard output sink, and writing of the file will resume from the point at which it had been suspended.

An alternative method of temporarily sending output to the console is to specify the console in calls on the output functions (see Section 5.8.4).

FILE-WRITE-POSITION [*n*, *filename*] Function

If *filename* is an open output file and *n* is zero or a positive integer, (FILE-WRITE-POSITION *n filename*) sets the write position of *filename* to *n* and returns T. If *n* is neither zero nor a positive integer, FILE-WRITE-POSITION returns the current write position of *filename*. If *filename* is not an open output file, FILE-WRITE-POSITION returns NIL. If no *filename* argument is given, the standard output file is assumed.

A file's **write position** determines the point in the file at which the next character will be written. When a file is opened for output, the file's write position is set to 0 (the first byte of the file). After a character is written to the file, the file's write position is automatically incremented. Thus, use FILE-WRITE-POSITION when a file needs to be written to non-sequentially or when it is necessary to reduce the size of a file (see CLOSE-OUTPUT-FILE). To save the entirety of a non-sequentially written file, remember to set the file's write position to the end of the file *before* closing the file.

FILE-WRITE-POSITION makes it possible to create files containing "holes" where no characters have been written. If an attempt is made to read characters from such holes, an End-Of-File error break occurs.

```
(DEFUN FILE-WRITE-POSITION (N FILE-NAME)
  ((NULL FILE-NAME)
    ((OUTPUT-FILE)
      (FILE-WRITE-POSITION N (OUTPUT-FILE)) ) )
  ((OUTPUT-FILE-P FILE-NAME)
    ((AND (INTEGERP N) (>= N 0))
      (set the write position of FILE-NAME to N) )
    (FILE-NAME's current write position) ) )

$ (OPEN-OUTPUT-FILE 'TEST1.LSP)                          ;Open TEST1.LSP
C:TEST1.LSP

$ (FILE-WRITE-POSITION)                                  ;Initial write position
0

$ (FILE-WRITE-POSITION 100)                              ;Set write position to 100
T

$ (FILE-WRITE-POSITION)                                  ;New write position
100
```

FILE-LINE-LENGTH [*n*, *filename*] Function

If *filename* is an open output file and *n* is a positive integer, (FILE-LINE-LENGTH *n* *filename*) sets the line length of *filename* to *n* and returns T. If *n* is not a positive integer, FILE-LINE-LENGTH returns the current line length of *filename*. If *filename* is not an open output file, FILE-LINE-LENGTH returns NIL. If no *filename* argument is given, the standard output file is assumed.

If *AUTO-NEWLINE* is nonNIL, newline characters are automatically sent to an open output file as necessary to enforce the file's line length. If *AUTO-NEWLINE* is NIL, the file's line length is ignored (see Section 5.8.5).

```
(DEFUN FILE-LINE-LENGTH (N FILE-NAME)
  ((NULL FILE-NAME)
    ((OUTPUT-FILE)
      (FILE-LINE-LENGTH N (OUTPUT-FILE)) ) )
  ((OUTPUT-FILE-P FILE-NAME)
    ((AND (INTEGERP N) (PLUSP N))
      (set the line length of FILE-NAME to N)
      T )
    (FILE-NAME's current line length) ) )

$ (OPEN-OUTPUT-FILE 'TEST1.LSP)                          ;Open TEST1.LSP
C:TEST1.LSP

$ (FILE-LINE-LENGTH)                                   ;Default line length
79

$ (FILE-LINE-LENGTH 120)                             ;Set line length to 120
T

$ (FILE-LINE-LENGTH)                                       ;New line length
120
```

5.9.3 File and Directory

The file functions enable programs to delete a file, rename a file, and determine a file's existence, date, and length. The directory functions enable programs to determine and/or modify the default drive and current directory, and produce a list of the files in a specified directory.

RENAME-FILE [*old-filename, new-filename*] Function
DELETE-FILE [*filename*] Function
PROBE-FILE [*filename*] Function

If the file on the drive and directory specified by *old-filename* exists, (RENAME-FILE *old-filename new-filename*) renames *old-filename* to *new-filename* and returns T. If no file specified by *old-filename* exists, RENAME-FILE returns NIL.

If the file on the drive and directory specified by *filename* exists, (DELETE-FILE *filename*) deletes *filename* and returns T. If no file specified by *filename* exists, DELETE-FILE returns NIL.

If the file on the drive and directory specified by *filename* exists, (PROBE-FILE *filename*) returns *filename*. If no file specified by *filename* exists, PROBE-FILE returns NIL.

```
$ (RENAME-FILE 'TEMP.LSP 'TEST.LSP)                    ;Rename file
T

$ (PROBE-FILE 'TEST.LSP)                              ;There is a TEST.LSP
TEST.LSP

$ (DELETE-FILE 'TEST.LSP)                               ;Delete TEST.LSP
T

$ (PROBE-FILE 'TEST.LSP)                             ;There is no TEST.LSP
NIL
```

FILE-DATE [*filename*] Function
FILE-LENGTH [*filename*] Function

If the file on the drive and directory specified by *filename* exists, (FILE-DATE *filename*) returns the date *filename* was written as a list of six integers signifying the following: year (A.D.), month (1-12), day (1-31), hour (0-23), minute (0-59), and second (0-59). If no file specified by *filename* exists, FILE-DATE returns NIL.

If *filename* is an open input and/or output file, (FILE-LENGTH *filename*) returns the length of *filename* in bytes. If *filename* is not open, FILE-LENGTH returns NIL.

```
$ (FILE-DATE 'ANIMAL.LSP)                          ;Date of ANIMAL.LSP
(1986 12 23 10 43 4)                          ;December 23, 1986, 10:43:04

$ (OPEN-INPUT-FILE 'ANIMAL.LSP)                    ;Open ANIMAL.LSP
C:ANIMAL.LSP

$ (FILE-LENGTH 'ANIMAL.LSP)                        ;Length of ANIMAL.LSP
8454
```

DEFAULT-DRIVE [*drive*] Function
DEFAULT-PATH [*drive*, *path*] Function
DIRECTORY [*pathname*] Function

If *drive* is a valid drive letter, (DEFAULT-DRIVE *drive*) makes *drive* the default drive and returns the total number of drives. (DEFAULT-DRIVE) returns the drive letter of the default drive.

If *drive* is a valid drive letter and *path* is a valid directory path, (DEFAULT-PATH *drive path*) makes *path* the current directory on *drive* and returns T. (DEFAULT-PATH *drive*) returns the current directory on *drive*. (DEFAULT-PATH) returns the current directory on the default drive.

If *pathname* specifies a valid drive and directory, (DIRECTORY *pathname*) returns a list of the names of the files in that directory that match *pathname*. The **wild-card characters, ?** and *, can be used to specify multiple files (see the MS-DOS manual). (DIRECTORY) returns a list of all the files in the current directory on the default drive.

```
$ (DEFAULT-DRIVE)                                    ;Default drive is A
A

$ (DEFAULT-DRIVE 'C)                                 ;Change drive to C
2                                                     ;A 2 drive system

$ (DEFAULT-PATH)                                  ;Current directory is MULISP
MULISP

$ (DEFAULT-PATH 'C 'BIN)                           ;Change directory to C:\BIN
T

$ (DIRECTORY '|*.COM|)                         ;The COM files in current directory
(MULISP.COM COMTOEXE.COM)
```

GETSET [*parameter*] Function

If running under MS-DOS version 2.0 or later, (GETSET *parameter*) returns the symbol whose print name is the MS-DOS environment parameter associated with *parameter*. If the parameter is not found, GETSET returns NIL.

If running under MS-DOS version 3.0 or later, (GETSET) returns the drive, path, and file name of the program currently being executed, normally MULISP.COM. This makes it possible for **muLISP** application programs to determine the drive and directory from which they were launched. It is often convenient for application programs to make that drive and directory the default when searching for data and other auxiliary files.

```
$ (GETSET 'PATH)                                    ;Get MS-DOS search directories
|C:\\;C:\\BIN;C:\\MULISP|                        ;They are C:\, C:\BIN, and C:\MULISP

$ (GETSET 'COMSPEC)                                 ;Get MS-DOS command processor
|C:\\COMMAND.COM|                                        ;It is C:\COMMAND.COM

$ (GETSET)                                                        ;Get launch path
|C:\\MULISP\\MULISP.COM|                          ;It is C:\MULISP\MULISP.COM
```

5.9.4 Memory Image Files

The functions in this subsection are for saving and loading memory image files. These files are compact binary files containing the environment present at the time they are saved. The environment consists of all the data structures, variable values, property values, and function definitions in **muLISP**. Once a program has been debugged, its memory image file is usually faster and more convenient to load than its source files (see Section 2.6).

SAVE [*filename*] Function

If *filename* includes a file name extension *other* than COM, (SAVE *filename*) saves a memory image SYS file on the drive and directory specified by *filename* and returns T. SAVE always gives memory image files a file name extension of SYS. If there is no directory entry available or there is insufficient disk space for the memory image file, SAVE returns NIL.

If *filename* includes a file name extension of COM, (SAVE *filename*) saves a memory image COM file that includes the **muLISP** interpreter, minus the pseudo-code compiler, on the drive and directory specified by *filename* and returns T. The pseudo-code compiler can be preserved in COM files created by SAVE by modifying a byte in the **muLISP** base page (see Appendix H).

When saving a memory image file, all active data structures are automatically compacted so that the resulting file is of minimum size. Therefore, the size of the memory image file increases with the amount of active data structures present at the time of the save.

If a COM file is less than 65280 bytes, it can be executed from the MS-DOS command line. If a COM file is more than 65280 bytes, it must first be converted to an EXE file using the utility COMTOEXE.COM distributed with **muLISP**. If the COM file specified by *filename* exists, the MS-DOS command

COMTOEXE *filename drive*

creates an EXE file equivalent to the COM file on drive *drive* and then deletes the COM file. If *drive* is omitted, the EXE file is created on the default drive.

Since COM and EXE files are larger than SYS files, normally have their pseudo-code compiler disabled, and cannot be loaded into a running **muLISP** system, SYS files are normally used during program development. COM and EXE files, on the other hand, are primarily useful for packaging completed application programs.

Application programs saved as a COM or EXE file can be distributed to others provided a) the pseudo-code compiler is disabled, and b) you agree to the terms of the **muLISP** Runtime License Agreement and pay the required fee. For details, write **Soft Warehouse, Inc.**, 3660 Waialae Avenue, Suite 304, Honolulu, Hawaii, 96816-3236, U.S.A.

```
(DEFUN SAVE (FILE-NAME)
  (close all open input and output files)
  (RECLAIM)
  ((FILE-NAME specifies a COM file)
    (disable the muLISP pseudo-code compiler)
    (save a memory image COM file that includes the muLISP interpreter)
    T )
  (save a memory image file)
  T )

$ (SAVE 'B:WHALE)                                    ;Save WHALE.SYS on drive B:
T

$ (SAVE 'WHALE.COM)                                  ;Save WHALE.COM on default drive
T
```

LOAD [*filename, display*] Function
LOAD-VERBOSE: T Control variable

If *filename* includes the file name extension SYS and such a memory image SYS file exists, (LOAD *filename*) loads the file and begins execution of its top-level loop (see Section 5.10). The top-level loop in turn calls the function bound to the variable DRIVER in the SYS file. The remainder of the characters in *filename* after the SYS file name are passed as the command line to the top-level loop. LOAD does *not* return a value if the SYS file is successfully loaded, otherwise it returns NIL.

If *filename* includes a file name extension *other* than SYS and such a source file exists, (LOAD *filename*) loads *filename* and returns T. If *LOAD-VERBOSE* is nonNIL, LOAD displays the message

;Loading *filename*

If *display* is nonNIL, (LOAD *filename display*) displays the value of each expression after it is evaluated. This facilitates finding syntax errors in large source files. Examining the value of the variables +, ++, +++, *, **, and *** may also be useful in finding syntax errors (see Section 5.10.2).

If *filename* does not include a file name extension, LOAD searches for a memory image SYS file to load. If no such file exists, then LOAD searches for a source file with an extension name of LSP to load. If there is no memory image or source file named *filename*, then LOAD returns NIL.

```
(DEFUN LOAD (FILE-NAME DISPLAY)
  ((if FILE-NAME does not include a file name extension)
    ((LOAD FILE-NAME.SYS DISPLAY))
    (LOAD FILE-NAME.LSP DISPLAY) )
  ((if FILE-NAME includes a file name extension of SYS)
    (close all open input and output files)
    ((if the memory image file FILE-NAME exists)
      (load the memory image file FILE-NAME)
      (FUNCALL DRIVER) ) )
  ((if the source file FILE-NAME exists)
    (load the source file FILE-NAME)
    T ) )

$ (LOAD 'C:WHALE)                          ;Load WHALE.SYS from drive C:

$ (LOAD 'COMMON)                          ;Read in and evaluate COMMON.LSP
T
```

5.10 User Interface

The primitives in this section provide the basic user interface to **muLISP**.

5.10.1 Top-level Loop

When **muLISP** starts it initializes its data spaces and enters the **top-level loop.** This loop sets *BREAK-LEVEL* to 0 and then calls the current **driver function** by issuing the call (FUNCALL DRIVER *command-line*) where *command-line* is the MS-DOS command line that was used to start **muLISP**. The command line can be used to pass a file name to a program's driver function. For instance, the **muLISP** editor takes advantage of this feature.

Upon return from the driver function, the top-level loop catches all throws not previously caught by the application program or the current driver function, and deactivates the throw. If *THROW-TAG* is the symbol SYSTEM, the top-level loop closes all open files, restores the normal cursor shape on PC compatible computers, terminates **muLISP**, and returns control to MS-DOS. If *THROW-TAG* is the symbol Disk Full or Memory Full, the top-level loop generates an appropriate error break.

```
(LOOP
  (SETQ *BREAK-LEVEL* 0)
  (CATCH NIL (FUNCALL DRIVER command-line))
  (SETQ command-line NIL)
  ( ((throw was not active?))
    ((EQ *THROW-TAG* 'SYSTEM)
      (close all currently open files)
      (restore the normal cursor shape)
      (terminate muLISP) )
    ((EQ *THROW-TAG* "Memory Full")
      (display GC statistics)
      (BREAK NIL "Memory Full") )
    ((EQ *THROW-TAG* "Disk Full")
      (BREAK NIL "Disk Full) ) ) )
```

DRIVER: 'DRIVER — Control variable

The initial value of the *control variable* DRIVER is the *function* DRIVER that implements the primitively defined read-eval-print loop (see Section 5.10.2). However, this loop can be replaced with a new driver function by setting the *control variable* DRIVER to the name of the new driver function. Then the command (RETURN) will terminate the read-eval-print loop and the top-level loop will automatically call the new driver function.

The following example shows how to implement and then activate a simple **eval-quote LISP** driver loop:

```
$ (DEFUN EVAL-QUOTE ()                                    ;Define eval-quote loop
    (CATCH 'RETURN (LOOP
      (CATCH 'DRIVER
        (TERPRI NIL T)
        (WRITE-STRING "> " T)
        (PRINT (APPLY (READ) (READ))) ))) )
EVAL-QUOTE

$ (SETQ DRIVER 'EVAL-QUOTE)                                ;Reassign DRIVER
EVAL-QUOTE

$ (RETURN)                                                 ;Exit from default loop

> CONS (DOG (CAT COW PIG))                                 ;Eval-quote interaction
(DOG CAT COW PIG)
```

5.10.2 Read-eval-print Loop

The primitively defined read-eval-print loop is a relatively unsophisticated interface for interacting with **muLISP**. Most programs replace it with an interface more suited for the application.

DRIVER [*filename*]	Function
BREAK-LEVEL	Variable
+	Variable
++	Variable
+++	Variable
*	Variable
**	Variable
***	Variable
-	Variable

DRIVER implements the standard **muLISP read-eval-print loop**. It is the default driver function called by the top-level loop. DRIVER is also called when an error or interrupt break occurs and the user selects the Break option.

If *filename* is nonNIL and names an existing memory image or source file, (DRIVER *filename*) attempts to load *filename* (see Section 5.9.4). If *filename* is nonNIL but no such file exists, DRIVER displays the File not found warning message (see Section 4.5). If *filename* is NIL, DRIVER goes straight to the following read-eval-print loop:

If *BREAK-LEVEL* is a positive integer (i.e. if DRIVER was not called directly by the top-level loop), DRIVER displays the value of *BREAK-LEVEL*, otherwise it is not displayed. Next it displays the **muLISP** prompt (a dollar sign ($) followed by a space). Finally it reads an expression from the console keyboard using the function READ, evaluates the expression using the function EVAL, and displays the result on the console screen using the function WRITE.

As a programming convenience, DRIVER assigns the last three expressions it reads to the variables +, ++, and +++ respectively. Also it assigns the result of the last three expressions it evaluates to the variables *, **, and *** respectively. Finally it assigns the expression it has read most recently to the variable - before it evaluates the expression. The example on the next page illustrates the use of these variables.

DRIVER catches throws to the tag DRIVER that may occur during the reading, evaluating, or printing of expressions. This feature makes it possible for functions such as BREAK to abort program execution and immediately pass control to the current level read-eval-print loop.

DRIVER repeats the read-eval-print loop until the user enters the expression (RETURN *form*) or presses the **Esc** key. After either of these actions, DRIVER terminates the loop, and returns the result of evaluating *form* in the former case or returns NIL in the latter case.

```
(DEFUN DRIVER (FILE-NAME)
  ( ((NULL FILE-NAME))                                          ;File name given?
    ((LOAD FILE-NAME))                                             ;Load file
    (WRITE-LINE "File not found" T) )                           ;File not found
  (LOOP
    (TERPRI NIL T)
    ( ((ZEROP *BREAK-LEVEL*))                                    ;Zero-level loop?
      (WRITE *BREAK-LEVEL* T) )                                 ;Display break level
    (WRITE-STRING "$ " T)                                        ;Display prompt
    (CATCH NIL
           (SETQ - (READ T))                                     ;Read expression
           (PSETQ * (UNWIND-PROTECT (EVAL -)                     ;Evaluate expression
                                    (SETQ +++ ++                 ;Update variables
                                    ++  +
                                    +   -) )
                  *** **
                  **  *)
           (WRITE * T)                                           ;Display result
           (TERPRI NIL T) )
    ( ((throw was not active?))
      ((EQ *THROW-TAG* 'DRIVER)                                ;Catch throws to DRIVER
        (deactivate throw) )
      ((EQ *THROW-TAG* 'RETURN)                                ;Catch throws to RETURN
        (deactivate throw)
        (RETURN *) )
      (THROW *THROW-TAG* *) ) ) )                              ;Continue other throws

$ (CONS 'SAM '(ANN JOE SUE))                                    ;Sample interaction
(SAM ANN JOE SUE)

$ +                                                             ;Get previous input
(CONS (QUOTE SAM) (QUOTE (ANN JOE SUE)))

$ **                                                            ;Get previous output
(SAM ANN JOE SUE)
```

5.10.3 Interrupt and Error Handling

BREAK [*object, message*] Function
BREAK Variable

If *message* is a nonNIL symbol, (BREAK *object message*) suspends program execution, displays *message*, Break:, and *object* on the console screen. If *message* is NIL or omitted, BREAK does not display a message or *object*. On the next line, BREAK displays the break **option prompt**

Continue, Break, Abort, Top-level, Restart, System?

Then BREAK waits for the user to select one of the options by entering the first letter of its name (C, B, A, T, R, or S). The options are listed in order of increasingly drastic action. If after a few seconds the user fails to select an option, a beep is sounded to inform the user of the occurrence of a break.

The following summarizes the effect of selecting each of the break options:

Continue: returns *object* as the value of the call on BREAK, and execution of the program continues normally.

Break: temporarily suspends program execution, sets the variable BREAK to *object*, and increments *BREAK-LEVEL*. The function DRIVER is called to permit the user to examine the environment present at the time the break occurred. Since *BREAK-LEVEL* is now a positive integer, DRIVER displays *BREAK-LEVEL* as part of the prompt to remind the user that a break is in progress.

When ready to exit the break, the user can issue the command (RETURN *form*). This command evaluates *form*, exits the break, and returns the result as the value of the break. For example, the following command evaluates and returns *object*:

(RETURN (EVAL BREAK))

Just pressing the **Esc** key also exits the break and returns NIL as the value of the break.

Upon return from the break, BREAK decrements *BREAK-LEVEL*, restores the variable BREAK to its original value, and returns the value of the break as the value of the call on BREAK.

Abort: aborts the evaluation in progress at the time of the break and throws control to the most recent catch labeled DRIVER (usually the read-eval-print loop).

Top-level: aborts the evaluation in progress at the time of the break and throws control to the most recent catch labeled NIL. If not caught by the program, the top-level loop catches the throw and then calls the current driver function (see Section 5.10.1).

Restart: closes all open files, abandons the current environment, and initiates a fresh **muLISP** system. This is done only after the user confirms the action by pressing Y in response to the prompt Are you sure (Y/N)? .

System: closes all open files, terminates execution of **muLISP**, and returns control to the operating system. This is done only after the user confirms the action by pressing Y in response to the prompt Are you sure (Y/N)? .

When an error condition is detected, BREAK is called with the errant function form and an appropriate error message letting the user decide how to proceed. BREAK can be redefined as necessary to suit the application.

Break-points are calls to the function BREAK in function definitions. When a break-point is encountered in the execution of a program, variable values and function definitions can be examined and/or changed before proceeding with the execution. This can greatly facilitate the debugging of programs. The **muLISP** debugger makes it easy to insert and remove break-points at the entry point of functions. Break-points can be inserted anywhere within function definitions using the **muLISP** editor.

```
(DEFUN BREAK (BREAK MSG
    CHAR )
  ( ((NULL MSG))
    (TERPRI NIL T)
    (PRINC MSG T)
    (WRITE-STRING " Break: " T)
    (WRITE BREAK T)
    (TERPRI NIL T) )
  (WRITE-STRING "Continue, Break, Abort, Top-level, Restart, System? " T)
  (LOOP
    (WRITE-BYTE 7 NIL T)
    (SETQ CHAR (CHAR-UPCASE (ASCII (READ-BYTE T))))
    ((EQ CHAR 'A)
      (THROW 'DRIVER BREAK) )
    ((EQ CHAR 'B)
      (INCQ *BREAK-LEVEL*)
      (CLEAR-INPUT)
      (SETQ BREAK (DRIVER))
      (DECQ *BREAK-LEVEL*)
      BREAK )
    ((EQ CHAR 'C)
      BREAK )
    ((EQ CHAR 'T)
      (THROW NIL BREAK) )
    ((EQ CHAR 'R)
      (RESTART) )
    ((EQ CHAR 'S)
      (SYSTEM) ) ) )
```

CONSOLE-INTERRUPT [ ] Function
INTERRUPT-HOOK: 'CONSOLE-INTERRUPT Control variable
UNDEFINED-FUNCTION [*form*] Function
UNDEFINED-HOOK: 'UNDEFINED-FUNCTION Control variable

(CONSOLE-INTERRUPT) generates a Console Interrupt break by evaluating the form (BREAK NIL "Console Interrupt"). At the break option prompt, the user can choose to abort the program or continue execution.

If *INTERRUPT-HOOK* is the name of a function and the **Esc** key is pressed during the execution of a program, the program is suspended while the function is called with no arguments. If *INTERRUPT-HOOK* is NIL, user initiated console interrupts are disabled and the **Esc** key is read just like any other character that has been entered from the keyboard. Otherwise, an Undefined Function error break occurs. The initial value of *INTERRUPT-HOOK* is CONSOLE-INTERRUPT.

(UNDEFINED-FUNCTION *form*) generates an Undefined Function error break by evaluating the form (BREAK *form* "Undefined Function"). During the break, the user can examine the variable BREAK to determine the symbol that is undefined and optionally return a value in place of the undefined function call.

If *UNDEFINED-HOOK* is the name of a function and an attempt is made to evaluate a form whose car is not a function or a LAMBDA body, the program is suspended while the function is called with the errant form as its argument. The initial value of *UNDEFINED-HOOK* is UNDEFINED-FUNCTION.

INTERRUPT-HOOK and *UNDEFINED-HOOK* make it possible for programs to customize the handling of console interrupts and undefined function errors appropriate for the application.

```
(DEFUN CONSOLE-BREAK ()
  (BREAK NIL "Console Interrupt") )

(DEFUN UNDEFINED-FUNCTION (FORM)
  (BREAK FORM "Undefined Function") )

$ (A B C)
Undefined Function Break: (A B C)
Continue, Break, Abort, Top-level, Restart, System? A
```

+TRAP [$object_1$, $object_2$] Function
-TRAP [$object_1$, $object_2$] Function
*TRAP [$object_1$, $object_2$] Function
/TRAP [$object_1$, $object_2$] Function
MAXTRAP [$object_1$, $object_2$] Function
MINTRAP [$object_1$, $object_2$] Function
GCDTRAP [$object_1$, $object_2$] Function
LCMTRAP [$object_1$, $object_2$] Function
ABSTRAP [*object*] Function
NUMTRAP [*object*] Function
DENTRAP [*object*] Function

Instead of generating an error break directly when a Nonnumeric Argument or Zero Divide error occurs, some numerical functions have their own error trap handlers to generate the error break.

By appropriately redefining these error traps, programs can modify the servicing of these errors as may be required by an application. For example a computer algebra system would want these functions to handle algebraic as well as numeric arguments.

```
(DEFUN +TRAP LST                                                    ;+
  (BREAK (CONS '+ LST) "Nonnumeric Argument") )

(DEFUN -TRAP LST                                                    ;-
  (BREAK (CONS '- LST) "Nonnumeric Argument") )

(DEFUN *TRAP LST                                                    ;*
  (BREAK (CONS '* LST) "Nonnumeric Argument") )

(DEFUN /TRAP LST                                                    ;/
  ((ZEROP (CADR LST))
    (BREAK (CONS '/ LST) "Zero Divide") )
  (BREAK (CONS '/ LST) "Nonnumeric Argument") )

(DEFUN MAXTRAP LST                                                  ;MAX
  (BREAK (CONS 'MAX LST) "Nonnumeric Argument") )

(DEFUN GCDTRAP LST                                                  ;GCD
  (BREAK (CONS 'GCD LST) "Nonnumeric Argument") )

(DEFUN ABSTRAP LST                                                  ;ABS
  (BREAK (CONS 'ABS LST) "Nonnumeric Argument") )

(DEFUN NUMTRAP LST                                                  ;NUMERATOR
  (BREAK (CONS 'NUMERATOR LST) "Nonnumeric Argument") )
```

5.10.4 User Querying

Y-OR-N-P [*message*] Library function
YES-OR-NO-P [*message*] Library function

If *message* is a nonNIL symbol, (Y-OR-N-P *message*) displays *message* on a fresh line immediately followed by the prompt (Y/N), clears the console's input buffer, and waits for the user to press Y or N (ignoring case). If the user presses Y, Y-OR-N-P returns T. If the user presses N, it returns NIL.

If *message* is a nonNIL symbol, (YES-OR-NO-P *message*) displays *message* on a fresh line immediately followed by the prompt (Yes or No), clears the console's line edit buffer, and waits for the user to type Yes or No (ignoring case) and press the **Enter** key. If the user enters Yes, YES-OR-NO-P returns T. If the user enters No, it returns NIL.

These predicates provide a standard convention for asking the user yes-or-no type questions. Y-OR-N-P should be used for non-dangerous situations where a simple Y or N is a sufficient answer. YES-OR-NO-P should be used for potentially dangerous situations where the user should stop and think before answering.

```
(DEFUN Y-OR-N-P (MSG
    CHAR )
  ( ((NULL MSG))
    (FRESH-LINE T)
    (WRITE-STRING (PACK* MSG " (Y/N) ") T) )
  (CLEAR-INPUT T)
  (LOOP
    (SETQ CHAR (CHAR-UPCASE (ASCII (READ-BYTE T))))
    ((EQ CHAR 'Y)
      (WRITE-STRING CHAR T)
      T )
    ((EQ CHAR 'N)
      (WRITE-STRING CHAR T)
      NIL )
    (WRITE-BYTE 7 NIL T) ) )

$ (Y-OR-N-P "Continue lesson?")
Continue lesson? (Y/N) Y
T

$ (YES-OR-NO-P "Delete your life's work?")
Delete your life's work? (Yes or No) NO
NIL
```

5.10.5 Line Editor

When reading a line of text from the console, **muLISP** provides the user with a powerful line editor with many control character commands (see Section 2.3). This line editor is called automatically by the input functions (see Section 5.8.1). However, the line editor can also be called directly, enabling a program to insert text into the line edit buffer and pre-position the cursor within the line of text. This makes it easy to present the user with default entries and to show the user the location of syntax errors.

LINE-EDIT [*text*, *point*, *column*, *width*] Function
LINE-TERMINATOR Variable
LINE-POINT Variable
LINE-COLUMN Variable

If *text* is an atom, *point* is zero or a positive integer less than or equal to the length of *text*, *width* is a positive integer less than or equal to the current window width minus the current cursor column position (see Section 5.8.6), and *column* is zero or a positive integer less than or equal to both *point* and *width*, (LINE-EDIT *text point column width*) displays *text* beginning from the current cursor position for a maximum of *width* columns, and positions the cursor at an offset of *column* columns from the beginning of the window. The portion of text displayed is determined such that the cursor is positioned on the *point*'th character in the text.

The default value of *text* is the empty string. The default value of *point* and *column* is zero. The default value of *width* is the current window width minus the current cursor column position.

After the user edits the line of text and presses the **Enter** key or any unrecognized control key or an extended function key on PC compatible computers, LINE-EDIT returns the line of text as a symbol. In addition, LINE-EDIT sets *LINE-POINT* to the final offset in the line of text, sets *LINE-COLUMN* to the final cursor column position, and sets *LINE-TERMINATOR* to the byte value of the key that terminated the line edit. If terminated by an extended function key, *LINE-TERMINATOR* is set to minus the key's extended function key code (see Appendix D).

Note that if *text* is the line of text returned by the last call to LINE-EDIT and *point* and *column* are the value of *LINE-POINT* and *LINE-COLUMN* respectively, LINE-EDIT resumes editing as if it had never been exited. Using *LINE-TERMINATOR*, programs can invoke functions based on the byte value of the key that terminated the line edit. Based on these facts, it is possible for programs to add new control character commands to those already provided by the line editor. It is also possible to redefine the primitive commands by modifying the Line Editor Control Key Table and the Extended Function Key Translation Table in the **muLISP** base page (see Appendix H).

INSERT-MODE: T Control variable

While line editing, if *INSERT-MODE* is nonNIL, new characters are added to the text rather than replacing old text. If *INSERT-MODE* is NIL, new characters replace old text. Thus, the line editor's **Ctrl-V** command simply sets *INSERT-MODE* to (NOT *INSERT-MODE*).

Given a default line of text, a column width, and an initial value for *INSERT-MODE*, the example function GET-LINE presents the user with the text in a one line window of the specified width. The user can edit the text using all the facilities provided by the **muLISP** line editor (see Section 2.3). If the user presses the **Enter** key, the editing is terminated and GET-LINE returns the newly edited line. If the user presses the **Esc** key, the editing is aborted and GET-LINE returns NIL.

If the user presses any non-recognized control or function key, GET-LINE sounds a warning beep before resuming the line-editing. Note that by testing for other values of *LINE-TERMINATOR*, GET-LINE can be extended to recognize additional function and/or control keys appropriate for a given application.

```
$ (DEFUN GET-LINE (LINE COLUMNS *INSERT-MODE*
    *LINE-POINT* *LINE-COLUMN* ROW COL)
  (SETQ ROW (ROW)
        COL (COLUMN))
  (LOOP                                                    ;Edit line
    (SET-CURSOR ROW COL)
    (SETQ LINE (LINE-EDIT LINE *LINE-POINT* *LINE-COLUMN* COLUMNS))
    ((EQ *LINE-TERMINATOR* 13)  LINE)                  ;Check for return
    ((EQ *LINE-TERMINATOR* 27)  NIL)                     ;Check for esc
    (WRITE-BYTE 7 NIL T) ) )                               ;Sound beep
```

Chapter 6. Source File Library

muLISP includes a library of source files that complement the facilities provided by the primitives described in Chapter 5. Files are provided that implement arrays, structures, irrational functions, flavors, dynamic closures, turtle graphics, and mouse interfaces.

Each source file contains all the required functions, macros, and variables necessary to implement the facilities described. Use a LOAD command to load a source file and make its facilities available for use (see Section 2.5).

This chapter describes the functions, macros, and variables provided by each source file. The presentation is similar to that used in Chapter 5, except that the formal definitions of functions and macros are *not* given. Instead you can use the **muLISP** Editor to examine the actual definitions in their source file.

6.1 Arrays

ARRAY.LSP provides a set of functions and macros for creating and accessing multi-dimensional arrays. This package is patterned after the Common LISP array facility (see Steele, 1984).

Arrays are realized as 3-element lists of the form

(ARRAY *contents dimensions*)

where *dimensions* is a list of the dimensions (i.e. the positive integer lengths of the axes) of the array, and *contents* is a tree whose list structure corresponds to the dimensions. Explicitly including dimension information in arrays provides for fast array bounds checking.

VECTOR [$object_1$, ..., $object_n$] Function
MAKE-ARRAY [*dimensions*, $option_1$, ..., $option_n$] Function

(VECTOR $object_1$ ... $object_n$) creates a **vector** (i.e. a one dimensional array) whose elements are the objects $object_1$ through $object_n$ respectively.

If *dimensions* is a list of zero or more positive integers, (MAKE-ARRAY *dimensions*) creates an array whose dimensions are the elements of *dimensions* and whose rank is the length of *dimensions*. If *dimensions* is a positive integer, (MAKE-ARRAY *dimensions*) creates a one-dimensional array whose dimension is *dimensions*. Note that the dimensions must be positive, whereas Common LISP only requires that they be non-negative. If *dimensions* is neither a positive integer nor a list of positive integers, a Dimension Error error break occurs. The following **array options** are supported:

:INITIAL-ELEMENT *expression* This option sets the initial value of each array element to *expression*. If this option is omitted, the initial value of each array element is NIL.

:INITIAL-CONTENTS *expression* This option sets the initial value of the array elements to the corresponding elements of *expression* which must be compatible in length with *dimensions* (i.e. the length of *expression* must be equal to the car of *dimensions* and each element of *expression* must be compatible in length with the cdr of *dimensions*). If not compatible in length, a Dimension Error error break occurs. If this option is omitted, the initial value of the array elements are determined by the :INITIAL-ELEMENT option.

```
$ (VECTOR 'MERCURY 'VENUS 'EARTH 'MARS)                    ;Create a vector
(ARRAY (MERCURY VENUS EARTH MARS) (4))

$ (MAKE-ARRAY '(2 3) :INITIAL-ELEMENT 5)                   ;Create an array
(ARRAY ((5 5 5) (5 5 5)) (2 3))

$ (MAKE-ARRAY '(2 3) :INITIAL-CONTENTS '((1 2 3) (4 5 6)))  ;Create an array
(ARRAY ((1 2 3) (4 5 6)) (2 3))
```

AREF [*array*, *subscript*$_1$, ..., *subscript*$_n$] Macro

If *array* is an *n*-dimensional array and *subscript*$_1$ through *subscript*$_n$ are non-negative integers, (AREF *array subscript*$_1$ ... *subscript*$_n$) returns the specified element of *array*. Note that the subscripts are zero based.

(SETF (AREF *array subscript*$_1$ ... *subscript*$_n$) *expression*) sets the specified element of array to *expression*.

For maximum efficiency, AREF does *not* check subscripts to insure that they are within the bounds of the array. If such checking is necessary, use the function ARRAY-IN-BOUNDS-P before calling AREF.

```
$ (SETQ PLANETS (VECTOR 'MERCURY 'VENUS 'EARTH 'MARS 'JUPITER, 'SATURN))
(ARRAY (MERCURY VENUS EARTH MARS JUPITER SATURN) (6))

$ (AREF PLANETS 2)                                          ;PLANETS[2]
EARTH

$ (AREF PLANETS 6)                                          ;PLANETS[6]
NIL

$ (SETF (AREF PLANETS 2) TERRA)                    ;Change PLANETS[2]
TERRA

$ (AREF PLANETS 2)                                          ;PLANETS[2]
TERRA
```

ARRAY-RANK [*array*] Function
ARRAY-DIMENSION [*array*, *axis*] Function
ARRAY-DIMENSIONS [*array*] Function
ARRAY-TOTAL-SIZE [*array*] Function

If *array* is an array, (ARRAY-RANK *array*) returns the **rank** (i.e. the number of dimensions or axes) of *array* as a non-negative integer. Otherwise, ARRAY-RANK returns NIL.

If *array* is an array and *axis* is a non-negative integer less than the rank of *array*, (ARRAY-DIMENSION *array axis*) returns the **dimension** (i.e. the positive integer length) of dimension number *axis* of *array*. Otherwise, ARRAY-DIMENSION returns NIL. Note that dimension number 0 is the first axis of *array*.

If *array* is an array, (ARRAY-DIMENSIONS *array*) returns the dimensions of *array* as a list of positive integers. Otherwise, ARRAY-DIMENSIONS returns NIL.

If *array* is an array, (ARRAY-TOTAL-SIZE *array*) returns the total number of elements of *array* (i.e. the product of all the dimensions). Otherwise, ARRAY-TOTAL-SIZE returns NIL.

```
$ (SETQ EG (MAKE-ARRAY '(2 3) :INITIAL-CONTENTS '((A B C) (D E F))))
(ARRAY ((A B C) (D E F)) (2 3))

$ (ARRAY-RANK EG)                                          ;Rank of array
2

$ (ARRAY-DIMENSION EG 1)                                 ;Dimension of array
3

$ (ARRAY-DIMENSIONS EG)                                 ;Dimensions of array
(2 3)

$ (ARRAY-TOTAL-SIZE EG)                                   ;Elements of array
6
```

ARRAYP [*object*] Function
ARRAY-IN-BOUNDS-P [*array*, *subscript*$_1$, ..., *subscript*$_n$] Function

If *object* is an array, (ARRAYP *object*) returns T, otherwise it returns NIL. Note that arrays are simply lists whose car is the symbol ARRAY.

If *n* is the rank of *array* and *subscript*$_1$ through *subscript*$_n$ are valid subscripts for the axes of *array*, (ARRAY-IN-BOUNDS-P *array subscript*$_1$ ... *subscript*$_n$) returns T, otherwise it returns NIL. *subscript*$_i$ is a valid subscript for the *i*th axis if it is a non-negative integer less than the dimension of the axis.

```
$ (SETQ EG (MAKE-ARRAY '(2 3) :INITIAL-CONTENTS '((A B C) (D E F))))
(ARRAY ((A B C) (D E F)) (2 3))

$ (ARRAYP EG)
T

$ (ARRAYP '(2 3 5))
NIL

$ (ARRAY-IN-BOUNDS-P EG 1 2)
T

$ (ARRAY-IN-BOUNDS-P EG 1 3)
NIL
```

6.2 Irrational Functions

IRRATNAL.LSP provides functions for computing exponential, logarithmic, trigonometric, and hyperbolic functions. If in exact mode and one of these functions needs to return an irrational number, an Exact Mode error break occurs.

EXACT-MODEP [ ] — Function
RATIONAL — Keyword

If the value of the control variable *EXACT-MODE* is neither NIL nor the symbol RATIONAL, (EXACT-MODEP) returns T, otherwise it returns NIL.

EXACT-MODEP is a predicate used to determine if exact mode arithmetic is in effect. Before a function in this subsection tries to compute an irrational number, it calls EXACT-MODEP to determine if approximate results are acceptable. If not, it generates an Exact Mode error break.

```
$ (EXACT-MODEP)                                    ;The initial mode is approximate
NIL
```

EXP [*n*] — Function

If *n* is a number, (EXP *n*) returns e^n. *e* is the base of the natural logarithms (2.71828...). EXP uses and updates if necessary the variable *E* to store *e* accurate to the current precision.

The result returned is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect and *n* is nonzero, an Exact Mode error break occurs.

```
$ (EXP 1)                                                    ;Compute e to 7 digits
2.7182818

$ (PRECISION 10)                                               ;Set precision to 10
1

$ (SETQ *PRINT-POINT* 60)                                        ;Print 60 digits
60

$ (EXP 1)                                                ;Compute e accurate to 52 digits
2.71828182845904523536028747135266249775724709369995955305672 2

$ (PRECISION 1)                                                ;Restore precision to 1
10

$ (SETQ *PRINT-POINT* 7)                                         ;Print 7 digits
7
```

EXPT [*n*, *m*] Function
SQRT [*n*] Function
ISQRT [*n*] Function

If *n* and *m* are numbers, (EXPT *n m*) returns n^m, that is, *n* raised to the *m*th power. (EXPT *n* 0) returns 1, even if *n* is 0.

If *m* is an integer or *n* is a perfect *m*th root, EXPT can return an exact result. Otherwise, the result returned by EXPT is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect and *m* is not an integer nor does *n* have a perfect *m*th root, an Exact Mode error break occurs. If *n* is negative and the denominator of *m* is even, an Invalid Argument error break occurs.

If *n* is a non-negative number, (SQRT *n*) returns the principal square root of *n*. The result is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect and *n* does not have a perfect square root, an Exact Mode error break occurs. If *n* is negative, an Invalid Argument error break occurs.

If *n* is a non-negative integer, (ISQRT *n*) returns the greatest integer less than or equal to the positive square root of *n*. ISQRT uses **Newton's method** to compute the square root. If *n* is negative, an Invalid Argument error break occurs.

```
(DEFUN SQRT (N)
  (EXPT N 1/2) )

$ (EXPT 2 3)                                          ;2 cubed
8

$ (EXPT 3 1/3)                                  ;The cube root of 3
1.4422495

$ (EXPT -8 1/3)                                ;The cube root of -8
-2

$ (EXPT 5 -3)                                        ;5 to the -3
0.008

$ (SQRT 5)                                    ;The square root of 5
2.2360679

$ (* * *)                                ;Square the previous answer
5

$ (ISQRT 5)                             ;The integer square root of 5
2

$ (ISQRT 1000000)                  ;The integer square root of 1000000
1000
```

LN [*n*] Function
LOG [*n*, *base*] Function

If *n* is a positive number, (LN *n*) and (LOG *n*) return the natural logarithm of *n*. If *n* and *base* are positive numbers, (LOG *n base*) returns the logarithm of *n* in the base *base*. *base* defaults to *e*, the base of the natural logarithms (2.71828...).

LN and LOG use and update if necessary the variable *LN2* to store the natural logarithm of 2, accurate to the current precision.

The result returned by LN and LOG is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect and *n* is not 1, an Exact Mode error break occurs. If *n* or *base* is *not* positive, an Invalid Argument error break occurs.

```
(DEFUN LOG (N M)
  ((NOT M)
    (LN N) )
  (/ (LN N) (LN M)) )

$ (LN 2)                                        ;The natural log of 2
0.6931471

$ (LOG 2 10)                                    ;The log of 2 to the base 10
0.3010299

$ (EXPT 10 *)                                   ;Raise 10 to the previous answer
2                                                ;Note exact result
```

PI [ ] Function

(PI) returns π, the ratio of the circumference to the diameter of a circle (3.14159...). PI uses and updates if necessary the variable *PI* to store π accurate to the current precision.

The result returned by PI is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an Exact Mode error break occurs.

In the following examples the *inaccurate* digits are underlined.

```
$ (PI)                                                    ;Compute pi to 7 digits
3.1415929

$ (PRECISION 18)                                          ;Set precision to 18
1

$ (SETQ *PRINT-POINT* 100)                                ;Print 100 digits
100

$ (PI)                                                    ;Compute pi accurate to 90 digits
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803
0348255430218296

$ (PRECISION 20)                                          ;Set precision to 20
18

$ (PI)                                                    ;Compute pi accurate to 98 digits
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803
0348253421170680

$ (PRECISION 1)                                           ;Restore precision to 1
18

$ (SETQ *PRINT-POINT* 7)                                  ;Print 7 digits
7
```

SIN [*n*] Function
COS [*n*] Function
TAN [*n*] Function
COT [*n*] Function
SEC [*n*] Function
CSC [*n*] Function

If *n* is an angle measured in radians, (SIN *n*) returns the **sine** of *n*, (COS *n*) returns the **cosine** of *n*, and (TAN *n*) returns the **tangent** of *n*, (COT *n*) returns the **cotangent** of *n*, (SEC *n*) returns the **secant** of *n*, and (CSC *n*) returns the **cosecant** of *n*.

The result returned by these functions is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an Exact Mode error break occurs.

```
(DEFUN TAN (N)
  (/ (SIN N) (COS N)) )

(DEFUN COT (N)
  (/ (COS N) (SIN N)) )

(DEFUN SEC (N)
  (/ (COS N)) )

(DEFUN CSC (N)
  (/ (SIN N)) )

$ (SIN 0.37)                                        ;Sine of 0.37 radians
0.3616154

$ (COS (/ (PI) 4))                                  ;Cosine of pi/4 radians (45°)
0.7071068

$ (TAN (/ (PI) 4))                                  ;Tangent of pi/4 radians (45°)
1                                  ;Result exact, thanks to floating slash arithmetic!

$ (COT 0.37)                                        ;Cotangent of 0.37 radians
2.5782288
```

ATAN [*n*, *m*] Function
ACOT [*n*, *m*] Function
ASIN [*n*] Function
ACOS [*n*] Function
ASEC [*n*] Function
ACSC [*n*] Function

If *n* is a number, (ATAN *n*) returns the principal **arctangent** of *n*, and (ACOT *n*) returns the principal **arccotangent** of *n*. The value of (ATAN *n*) is always between $-\pi/2$ and $\pi/2$ inclusive, and the value of (ACOT *n*) is always between 0 and π inclusive.

If *y* and *x* are numbers, (ATAN *y x*) and (ACOT *x y*) return the angle of the point (*x*, *y*) measured in radians counterclockwise from the positive *x*-axis. The value of (ATAN *y x*) and (ACOT *x y*) is always between $-\pi$ (exclusive) and π (inclusive).

If *n* is a number between -1 and 1 inclusive, (ASIN *n*) returns the principal **arcsine** of *n*, and (ACOS *n*) returns the principal **arccosine** of *n*. The value of ASIN is always between $-\pi/2$ and $\pi/2$ inclusive, and the value of ACOS is always between 0 and π inclusive. If *n* is not between -1 and 1, an Invalid Argument error break occurs.

If *n* is a number whose absolute value is greater than or equal to 1, (ASEC *n*) returns the principal **arcsecant** of *n*, and (ACSC *n*) returns the principal **arccosecant** of *n*. The value of ASEC is always between 0 and π inclusive, and the value of ACSC is always between $-\pi/2$ an $\pi/2$ inclusive. If the absolute value of *n* is less than 1, an Invalid Argument error break occurs.

The result returned by these functions is an angle measured in radians as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an Exact Mode error break occurs.

```
(DEFUN ACOT (N M)
  ((NOT M)
    (- (/ (PI) 2) (ATAN N)) )
  (ATAN M N) )

(DEFUN ASIN (N)
  (ATAN (/ N (SQRT (- 1 (* N N))))) )

(DEFUN ACSC (N)
  (ASIN (/ N)) )

$ (ATAN -2/3)                          ;The angle whose tangent is -2/3
-0.5880026

$ (ATAN 2 -3)                          ;The angle of the point (2, -3)
2.5535903
```

SINH [*n*] Function
COSH [*n*] Function
TANH [*n*] Function
COTH [*n*] Function
SECH [*n*] Function
CSCH [*n*] Function

If *n* is a number, (SINH *n*) returns the **hyperbolic sine** of *n*, (COSH *n*) returns the **hyperbolic cosine** of *n*, (TANH *n*) returns the **hyperbolic tangent** of *n*, (COTH *n*) returns the **hyperbolic cotangent** of *n*, (SECH *n*) returns the **hyperbolic secant** of *n*, and (CSCH *n*) returns the **hyperbolic cosecant** of *n*.

The result returned by these functions is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an Exact Mode error break occurs.

```
(DEFUN SINH (N)
  (SETQ N (EXP N))
  (/ (- N (/ N)) 2) )

(DEFUN COSH (N)
  (SETQ N (EXP N))
  (/ (+ N (/ N)) 2) )

(DEFUN TANH (N)
  (/ (SINH N) (COSH N)) )

(DEFUN COTH (N)
  (/ (COSH N) (SINH N)) )
(DEFUN SECH (N)
  (/ (COSH N)) )

(DEFUN CSCH (N)
  (/ (SINH N)) )

$ (SINH 0.37)                          ;Hyperbolic sine of 0.37
0.3785001

$ (TANH 0.37)                          ;Hyperbolic tangent of 0.37
0.3539917
```

ASINH [*n*] Function
ACOSH [*n*] Function
ATANH [*n*] Function
ACOTH [*n*] Function
ASECH [*n*] Function
ACSCH [*n*] Function

If *n* is a number, (ASINH *n*) returns the **hyperbolic arcsine** of *n*, (ACOSH *n*) returns the **hyperbolic arccosine** of *n*, (ATANH *n*) returns the **hyperbolic arctangent** of *n*, (ACOTH *n*) returns the **hyperbolic arccotangent** of *n*, (ASECH *n*) returns the **hyperbolic arcsecant** of *n*, and (ACSCH *n*) returns the **hyperbolic arccosecant** of *n*.

The result returned by these functions is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an Exact Mode error break occurs.

```
(DEFUN ASINH (N)
  (LN (+ N (SQRT (+ (* N N) 1)))) )

(DEFUN ACOSH (N)
  (LN (+ (ABS N) (SQRT (- (* N N) 1)))) )

(DEFUN ATANH (N)
  (/ (LN (/ (+ 1 N) (- 1 N))) 2) )

(DEFUN ACOTH (N)
  (/ (LN (/ (+ N 1) (- N 1))) 2) )

(DEFUN ASECH (N)
  (ACOSH (/ N)) )

(DEFUN ACSCH (N)
  (ASINH (/ N)) )

$ (ASINH 0.37)                          ;Hyperbolic arcsine of 0.37
0.3620391

$ (ATANH 0.37)                          ;Hyperbolic arctangent of 0.37
0.3884231
```

6.3 Structures

STRUCTUR.LSP provides a macro for creating named and unnamed **record structures** (i.e. **data types**) with named components. When a structure is defined, a full set of constructors, accessors, recognizers, copiers, and assignment constructs are created for the new data type. This package is patterned after the Common LISP structure facility (see Steele, 1984).

A named record structure is realized as a list of the form

$$(name\ slot_1\ slot_2 \ldots\ slot_n)$$

and an unnamed record structure is realized as a list of the form

$$(slot_1\ slot_2 \ldots\ slot_n)$$

All of a structure's functions (i.e. its constructors, accessors, copiers, and predicates) are implemented as macros. This gives good execution speed and integrates nicely with **muLISP**'s SETF implementation. Thus, to achieve maximum efficiency, a structure should be defined *before* defining functions that call the structure's functions, so that the macro expansion can occur at compile time rather than at run time. During program development, however, the control variable MACROEXPAND should be set to NIL to prevent macro expansion until run time so structures can be redefined (see Section 5.7.5).

To achieve maximum efficiency, the structure functions do *not* perform compile time or run time error checking. If such checking is necessary, it can done by the functions that call the structure's functions.

The components of a structure definition (i.e. its options, slot names, slot values, a :NAMED flag, and structure functions) are stored on the property list of the structure's name. In addition, a list of the structures supported is stored on the property list of each structure function.

DEFSTRUCT [$(name\ option_1 \ldots\ option_m), slot_1, \ldots, slot_n$] Macro

If $option_1$ through $option_m$ are DEFSTRUCT options and $slot_1$ through $slot_n$ are slot descriptions, (DEFSTRUCT ($name\ option_1 \ldots\ option_m$) $slot_1 \ldots\ slot_n$) creates a new structure named *name*. If only the default defstruct options are required, (DEFSTRUCT $name\ slot_1 \ldots\ slot_n$) can be used instead. Each **slot description** has the form *slotname* or (*slotname form*), where *slotname* is a symbol and *form* is a form. Note that slot options are not supported.

If a nonsymbol is given as the name for a structure or a slot in the structure, a Nonsymbolic Argument error break occurs. If the definition for a function to be created by DEFSTRUCT is not equal to the existing definition of the function, an **Incompatible Definition** error break occurs. If an unsupported DEFSTRUCT option is given, an **Unsupported Option** error break occurs. The following **defstruct options** are supported:

:CONC-NAME This option determines the prefix to be concatenated onto the front of the structure's slot names to make the names of the structure's accessor functions. If this option is given with the nonNIL symbol *prefix* as its argument, *prefix* is used as the accessor function prefix. If this option is not given or is given without arguments, the structure's name followed by a hyphen is used as the prefix. If this option is given with NIL as its argument, no prefix is used and the slot names themselves are used for the names of the accessor functions.

:CONSTRUCTOR This option can be included more than once in the option list to make and determine the name of the structure's keyword-driven constructor function, and to make and determine the name of positional constructor functions for the structure.

If this option is given with the nonNIL symbol *cons-name* as its only argument, *cons-name* is used as the name of the structure's keyword-driven constructor function. If this option is not given or is given without arguments, (PACK* "MAKE-" *name*) is used as the name of the keyword-driven constructor function. If this option is given with NIL as its only argument, no keyword-driven constructor function is provided.

If this option is given with the symbol *cons-name* and the list *arglist*, which is a list of the structure's slot names, as its two arguments, a positional constructor function named *cons-name* is provided. The slots of the structure *cons-name* creates are initialized to the arguments of *cons-name* according to the position of the slots in *arglist*.

:COPIER This option makes and determines the name of the structure's copier function. If this option is given with the nonNIL symbol *copier-name* as its argument, *copier-name* is used as the name of the structure's copier function. If this option is not given or is given without arguments, (PACK* "COPY-" *name*) is used as the name of the copier function. If this option is given with NIL as its argument, no copier function is provided.

:PREDICATE This option makes and determines the name of the type predicate for named structures. If this option is given with the nonNIL symbol *predicate-name* as its argument, *predicate-name* is used as the name of the structure's type predicate. If this option is not given or is given without arguments, (PACK* *name* "-P") is used as the name of the type predicate. If this option is given with NIL as its argument, no type predicate is provided. If this option is given for an unnamed structure, an Unnamed Structure error break occurs.

:INCLUDE This option includes in the structure being defined the slots of a previously defined structure. In other words, the structure being defined becomes an extension and specialization of the included structure. Note that included structures must be named or unnamed consistent with the structure being defined.

If *structure* is a previously defined structure, (:INCLUDE *structure*) includes the slots of *structure* in the new structure being defined. If $slot_1$ through $slot_n$ are slot descriptions in *structure*, (:INCLUDE *structure* $slot_1$ ... $slot_n$) includes the slots of *structure* in the

structure being defined, but with the original default values overridden by $slot_1$ through $slot_n$.

If the name of an undefined structure is given for inclusion in a structure, an Undefined Structure error break occurs. If a named structure is given for inclusion in an unnamed structure, or an unnamed structure is given for inclusion in a named structure, an Include Error error break occurs.

:TYPE This option, used in conjunction with the :NAMED option, forces a particular representation for the structure. (:TYPE LIST) causes the structure to be represented as an *unnamed* list, unless the :NAMED option is also used. Note that only the structure type LIST is supported. If this option is omitted, the structure will be represented as a *named* list.

:NAMED This option forces the structure to be named irrespective of any :TYPE specifications. If this option is omitted, the structure will be named unless an explicit (:TYPE LIST) option is specified.

```
$ (DEFSTRUCT PERSON AGE  SEX  WEIGHT  HEIGHT)                    ;Define persons
PERSON
$ (SETQ AL (MAKE-PERSON))                                        ;Make AL a person
$ (SETF (PERSON-AGE AL) 40)                                      ;Al's age
40
$ (DEFSTRUCT (PROGRAMMER (:INCLUDE PERSON)) LANGUAGES COMPUTERS)  ;Define programmers
PROGRAMMER
$ (SETQ SAM (MAKE-PROGRAMMER))                                   ;Make SAM a programmer
$ (SETF (PROGRAMMER-LANGUAGES SAM) '(PASCAL C LISP))             ;Sam's languages
(PASCAL C LISP)
$ (SETF (PERSON-SEX SAM) 'MALE)                                  ;Assign Sam's sex
MALE
```

6.4 Flavors

FLAVORS.LSP provides a set of functions and macros to facilitate object-oriented programming. An attempt was made to optimize the speed without using excessive memory. Property lists are used to store all the methods and instance variables for the flavors and objects. Most of the common flavors options are included in the package and they follow the convention used on LISP machines. More detailed documentation on flavors can be found in the reference manuals for LISP machines or in Chapter 2 of Hank Bromley's **LISP Lore: A Guide to Programming the LISP Machine** published by Kluwer Academic Publishers.

A **flavor** is the general classification of a group of objects. An **object** is an instance of a flavor. Each flavor has **methods** that it can respond to and **instance variables** that can have values. Each object of a specified flavor has the same instance variables although the values associated with each variable may be different. The methods are functions that are called when a message of the same name is sent to the object. Within a method definition variables are locally bound, but only within the context of that method. They do not remain bound when the immediate environment of the method is changed. Note that this **lexical binding** of variables is an alternative to the **dynamic binding** of variables used elsewhere by **muLISP**.

Flavors can inherit other flavors, along with the methods and instance variables of those inherited flavors. The inherited flavors can also be controlled to some degree by the DEFFLAVOR options :INCLUDED-FLAVORS, :REQUIRED-FLAVORS, and :NO-VANILLA-FLAVOR. The :VANILLA-FLAVOR is included by default in all flavors. It has two methods attached to it although more can be added to conform to the LISP machine environment. The two methods included in this package are the :DESCRIBE and the :WHICH-OPERATIONS methods. When a :DESCRIBE message is sent to an object, a description of the object is displayed on the screen. The :WHICH-OPERATIONS message returns a list of all the method names to which the object will respond.

When a message is sent to an object, a **method combination** is constructed and then evaluated. The default method combination is the DAEMON combination. The inherited flavors are searched top-down, depth-first for the first method that matches the message. When a matching method is found, it is evaluated and the value returned. All :BEFORE daemons are evaluated before the primary method in the order that the flavors were combined, and all the :AFTER daemons are evaluated in reverse order.

Whoppers are methods that wrap around the primary method combination. All :BEFORE, :AFTER, and primary methods are always surrounded by any whoppers that have been defined for that method, with the whopper for the most recent flavor on the inherited flavors list on the outside.

Additionally :AND and :OR method combinations can be declared for each method. An :AND combination gets all the methods that match the message name for an object. These methods are then evaluated in order until one returns NIL or all return a nonNIL value. The :OR combination option works in a similar fashion but returns the first

nonNIL value or NIL if all methods return NIL. These options are declared in the DEFFLAVOR function.

When FLAVORS.LSP is loaded and if the control variable *FLAVOR-DEMO* is nonNIL, a simple window management system written using the Flavors Package is also loaded. The commented source listing for the window manager shows how to define flavors and methods, and how to create instances of flavors.

DEFFLAVOR [*name*, *varlist*, *flavorlist*, *option*$_1$, ..., *option*$_n$] Macro

(DEFFLAVOR name *varlist flavorlist option*$_1$ ... *option*$_n$) defines a new flavor named *name* and adds *name* to the global variable *ALL-FLAVOR-NAMES*. *varlist* is a list, each element of which is a variable or a cons of a variable and a list of the default value of that variable. *flavorlist* is a list of all the inherited flavors for the flavor *name*. The following **defflavor options** are supported:

:INITABLE-INSTANCE-VARIABLES This option makes the instance variables of the flavor **initable** (i.e. they can be initialized by MAKE-INSTANCE through the use of a keyword). This option is only required for completeness since all instance variables are initable by default.

:GETTABLE-INSTANCE-VARIABLES By itself, this option creates methods for getting the value of *all* the flavor's instance variables. (:GETTABLE-INSTANCE-VARIABLES *var*$_1$ ... *var*$_n$) creates methods for getting the value of only the instance variables *var*$_1$ through *var*$_n$. The name of the retrieval method for the instance variable *var*$_i$ will be :*var*$_i$.

:SETTABLE-INSTANCE-VARIABLES By itself, this option creates methods for setting the values of *all* the flavor's instance variables. (:SETTABLE-INSTANCE-VARIABLES *var*$_1$ ... *var*$_n$) creates methods for setting the value of only the instance variables *var*$_1$ through *var*$_n$. The name of the assignment method for the instance variable *var*$_i$ will be :SET-*var*$_i$.

:REQUIRED-FLAVORS (:REQUIRED-FLAVORS *flavor*$_1$ ... *flavor*$_n$) specifies that any flavor that includes the flavor *name* must also include the flavors *flavor*$_1$ through *flavor*$_n$. If an attempt is made to create an instance of *name* that does not include these flavors, a Missing Required Flavor error break occurs.

:INCLUDED-FLAVORS (:INCLUDED-FLAVORS *flavor*$_1$ ... *flavor*$_n$) specifies that any flavor that includes the flavor *name* must also include the flavors *flavor*$_1$ through *flavor*$_n$. If an attempt is made to create an instance of *name* that does not include

flavor$_i$, then *flavor*$_i$ is added to the inherited flavors of the instance immediately following the last flavor that has *flavor*$_i$ on its list of included flavors.

:METHOD-COMBINATION (:METHOD-COMBINATION (*type order method*$_1$ ... *method*$_n$) ...) declares the way that methods from different flavors are combined when the method is called. *type* can be either :DAEMON (default), :AND, or :OR. *order* is used for the :AND or :OR type and can be either :BASE-FLAVOR-FIRST or :BASE-FLAVOR-LAST.

:NO-VANILLA-FLAVOR This option excludes the primitive methods :DESCRIBE and :WHICH-OPERATIONS for instances of the flavor *name*. This option is rarely used.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT) C D) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFFLAVOR BAR () (FOO))
BAR

$ (SETQ THING (MAKE-INSTANCE 'BAR))
BAR-I1

$ (SEND THING :B)
CAT

$ (SEND THING :SET-B 5)
5

$ (SEND THING :B)
5
```

DEFMETHOD [(*flavor type method*), *arglist*, $form_1$, ..., $form_n$] Macro
UNDEFMETHOD [(*flavor type method*)] Macro

If *flavor* is the name of a flavor, (DEFMETHOD (*flavor type method*) *arglist* $form_1$ ... $form_n$) defines the new method *method* for *flavor* of type *type*. *arglist* is a list of variables. *type* is an optional argument that can be either :BEFORE or :AFTER. A method's type determines the order in which the combined methods are evaluated for a given method combination.

When the method is called, the forms are evaluated using an implicit PROGN in the same way as the body of a normal function. Within the forms of the method, instance variables can be referred to directly and SETQ can even be used to change the value of a variable. Note that changing the value of a variable using SETQ does *not* cause :BEFORE and :AFTER daemons to be evaluated. This is true even if they would be evaluated if the value were changed with a :SET-*var* method.

(UNDEFMETHOD (*flavor type method*)) removes the definition of *flavor*'s method *method* of type *type*.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT)) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFMETHOD (FOO :BAR) () (LIST A B))
(:METHOD FOO :BAR)

$ (SETQ THING (MAKE-INSTANCE 'FOO))
FOO-I1

$ (SEND THING :BAR)
(4 CAT)

$ (UNDEFMETHOD (FOO :BAR))
T

$ (SEND THING :BAR)
Unclaimed Message Break: (FOO-I1 :BAR)
```

DEFWHOPPER [(*flavor method*), *arglist*, $form_1$, ..., $form_n$] Macro

If *flavor* is the name of a flavor, (DEFWHOPPER (*flavor method*) *arglist* $form_1$ ... $form_n$) defines the new whopper *method* for *flavor*.

Whoppers are methods that wrap around the primary method combination. The primary method combination is substituted wherever the function CONTINUE-WHOPPER is called. DEFWHOPPER is similar in all other respects to DEFMETHOD.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT)) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFMETHOD (FOO :BAR) (X) (LIST A X))
(:METHOD FOO :BAR)

$ (SETQ THING (MAKE-INSTANCE 'FOO))
FOO-11

$ (SEND THING :BAR 5)
(4 5)

$ (DEFWHOPPER (FOO :BAR) (X) (CONTINUE-WHOPPER (* 10 X)))
(:METHOD FOO :AROUND :BAR)

$ (SEND THING :BAR 5)
(4 50)
```

MAKE-INSTANCE [*flavor, varlist*] Macro

(MAKE-INSTANCE *flavor varlist*) creates an instance of *flavor* and returns it as the symbol formed by concatenating the symbol flavor, the string "-I" and the incremented value of the global variable *INSTANCE-COUNT*. *varlist* consists of alternating keywords and values of the instance variables for *flavor*. After the object has been created it is sent an :init message.

If *flavor* does not include the required flavors, a Missing Required Flavor error break occurs. If not all the keywords in *varlist* are handled by the inherited and included flavors of *flavor*, a Keyword Not Handled error break occurs.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT)) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFMETHOD (FOO :BAR) () (LIST A B))
(:METHOD FOO :BAR)

$ (SETQ THING (MAKE-INSTANCE 'FOO))
FOO-I1

$ (SEND THING :B)
CAT
```

SEND [*object*, *operation*, arg_1, ..., arg_n] Macro
SELF Variable

(SEND *object operation* arg_1 ... arg_n) sends *object* the message or method *operation* with the arguments arg_1 through arg_n. SEND returns the result of the primary operation or the result specified by the method combination given when the *object*'s flavor was defined by DEFFLAVOR. The variable SELF is bound to *object* while the method is being evaluated so the method can manipulate *object* itself.

If *object* has no methods to handle *operation*, an Unclaimed Message error break occurs.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT)) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFMETHOD (FOO :BAR) () (LIST A B))
(:METHOD FOO :BAR)

$ (SETQ THING (MAKE-INSTANCE 'FOO))
FOO-11

$ (SEND THING :BAR)
(4 CAT)

$ (SEND THING :SET-A 3)
3

$ (SEND THING :NONSENSE)
Unclaimed Message Break: (FOO-11 :NONSENSE)
Continue, Break, Abort, Top-level, Restart, System
```

6.5 Closures

CLOSURE.LSP provides a set of functions for creating and manipulating dynamic closures in **muLISP**. This package is patterned after the Lisp Machine LISP dynamic closure package.

CLOSURE [*varlist, function*] Function

If *varlist* is a list of variables and *function* is a lambda expression or the name of a function, (CLOSURE *varlist function*) creates and returns a **dynamic closure** of *function* that saves the current values of the variables in *varlist*. Normally a dynamic closure created by CLOSURE is assigned to a variable so the closure can be called using FUNCALL or APPLY.

When a dynamic closure is called, it temporarily binds the variables in *varlist* to the values saved in the closure. Next, it applies *function* to the variables in *varlist*. Finally, it saves the new values of the variables after the function application, restores the variables to their original value, and returns the value of the function application.

Note that dynamic closures can be created manually by a user at the **muLISP** command level or automatically by a function within a program.

A **generator** is a special kind of function that produces the successive elements of a sequence. For instance, generators can be defined to produce random numbers or the elements of a sequence of numbers. In order to "know" what element of the sequence to produce next, a generator must keep information about its current state. Implementing a generator as a dynamic closure keeps the state information *local* to the generator. This avoids the possibility of name conflicts that can arise if global variables are used to store the state information.

The dynamic closure FIB generates the successive elements of the Fibonacci Series defined as follows:

$$F(0) = 1$$
$$F(1) = 1$$
$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$

```
$ (SETQ F1 1  F2 1)
1

$ (SETQ FIB (CLOSURE '(F1 F2) '(LAMBDA ()
              (PSETQ F2 F1  F1 (+ F1 F2))  F1) ))

$ (FUNCALL FIB)                                  ;Generate the Fibonacci Numbers
2
```

```
$ (FUNCALL FIB)
3

$ (FUNCALL FIB)
5

$ (FUNCALL FIB)
8

$ (FUNCALL FIB)
13
```

LET-CLOSED [*letlist, function*] Macro

If *letlist* is a list of *m* elements of the form (var_i $value_i$) and *function* is the name of a function or a lambda expression, (LET-CLOSED *letlist function*) evaluates the expressions $value_1$ through $value_m$ in order, and temporarily binds the resulting values to the variables var_1 through var_m. Next LET-CLOSED creates a dynamic closure of *function* over the variables var_1 through var_m with initial values $value_1$ through $value_m$, restores the original values to the variables var_1 through var_m, and returns the dynamic closure. If an element of *letlist* is just the symbol *var*, it is equivalent to the form (*var* NIL) being an element of *letlist*.

Often LET-CLOSED is more convenient than CLOSURE for creating dynamic closures since LET-CLOSED temporarily binds closure variables to an initial value when creating the closure.

```
$ (SETQ FIB (LET-CLOSED ((F1 1) (F2 1))
      '(LAMBDA () (PSETQ F2 F1  F1 (+ F1 F2))  F1) ))

$ (FUNCALL FIB)                                    ;Generate the Fibonacci Numbers
2

$ (FUNCALL FIB)
3

$ (FUNCALL FIB)
5

$ (FUNCALL FIB)
8

$ (FUNCALL FIB)
13
```

COPY-CLOSURE [*closure*] Function
CLOSUREP [*object*] Function

If *closure* is a dynamic closure, (COPY-CLOSURE *closure*) creates and returns a copy of *closure*. The copy has the same closure function and variables as *closure* and initially it has the same closure values as *closure*, but it maintains its own list of closure values separate from *closure*'s. Thus, the copy can operate independently of *closure*.

If *object* is a dynamic closure, (CLOSUREP *object*) returns T, otherwise it returns NIL.

```
$ (SETQ FIB1 (LET-CLOSED ((F1 1) (F2 1))
      '(LAMBDA () (PSETQ F2 F1  F1 (+ F1 F2))  F1) ))

$ (FUNCALL FIB1)                                          ;Call FIB1
2

$ (SETQ FIB2 (COPY-CLOSURE FIB1) NIL)                     ;Copy FIB1
NIL

$ (FUNCALL FIB2)                                          ;Call FIB2
3

$ (FUNCALL FIB2)                                          ;Call FIB2
5

$ (FUNCALL FIB1)                                    ;FIB1 still at 3
3

$ (CLOSUREP FIB2)                                   ;FIB2 is a closure
T
```

SYMEVAL-IN-CLOSURE [*closure, symbol*] Function
SET-IN-CLOSURE [*closure, symbol, object*] Function
LOCATE-IN-CLOSURE [*closure, symbol*] Function

If *closure* is a dynamic closure and *symbol* is a closure variable of *closure*, (SYMEVAL-IN-CLOSURE *closure symbol*) returns the value of *symbol* in *closure*. If *symbol* is not a closure variable of *closure*, SYMEVAL-IN-CLOSURE returns the current value of *symbol*.

If *closure* is a dynamic closure and *symbol* is a closure variable of *closure*, (SET-IN-CLOSURE *closure symbol object*) sets the value of *symbol* in *closure* to *object* and returns *object*. If *symbol* is not a closure variable of *closure*, SET-IN-CLOSURE sets the value of *symbol* to *object* and returns *object*.

If *closure* is a dynamic closure and *symbol* is a closure variable of *closure*, (LOCATE-IN-CLOSURE *closure symbol*) returns the alist pair for *symbol* in *closure*. If *symbol* is not a closure variable of *closure*, LOCATE-IN-CLOSURE returns *symbol*.

```
$ (SETQ F1 1  F2 1)
1

$ (SETQ FIB (CLOSURE '(F1 F2) '(LAMBDA ()
              (PSETQ F2 F1  F1 (+ F1 F2))  F1) ))

$ (FUNCALL FIB)                                          ;Call FIB
2

$ F1                                                ;Value of F1 outside FIB
1

$ (SYMEVAL-IN-CLOSURE FIB 'F1)                       ;Value of F1 inside FIB
2

$ (SET-IN-CLOSURE FIB 'F1 5)                            ;Set F1 to 5 in FIB
5

$ (FUNCALL FIB)                                                  ;Call FIB
6

$ (SYMEVAL-IN-CLOSURE FIB 'F1)                       ;Value of F1 inside FIB
6

$ (LOCATE-IN-CLOSURE FIB 'F1)                      ;Alist pair for F1 in FIB
(F1 . 6)
```

CLOSURE-ALIST [*closure*] Function
CLOSURE-FUNCTION [*closure*] Function
CLOSURE-VARIABLES [*closure*] Function
BOUNDP-IN-CLOSURE [*closure*, *symbol*] Function

If *closure* is a dynamic closure, (CLOSURE-ALIST *closure*) returns a list of the variables of *closure* paired with their closure values. This list is in the form of an association list (a-list) that is suitable for use by the functions ASSOC and RASSOC (see Section 5.2.8).

If *closure* is a dynamic closure, (CLOSURE-FUNCTION *closure*) returns its closure function (i.e. the function name or lambda expression that was given when *closure* was created).

If *closure* is a dynamic closure, (CLOSURE-VARIABLES *closure*) returns a list of its closure variables (i.e. a *copy* of the variable list that was given when *closure* was created).

If *closure* is a dynamic closure and *symbol* is a closure variable of *closure* (i.e. bound in *closure*), (BOUNDP-IN-CLOSURE *closure symbol*) returns T, otherwise it returns NIL.

```
$ (SETQ FIB (LET-CLOSED ((F1 1) (F2 1))
      '(LAMBDA () (PSETQ F2 F1  F1 (+ F1 F2))  F1) ))

$ (FUNCALL FIB)                                              ;Call FIB
2

$ (CLOSURE-ALIST FIB)                                        ;FIB a-list
((F1 . 2) (F2 . 1))

$ (CLOSURE-FUNCTION FIB)                                     ;FIB function
(LAMBDA NIL (PSETQ F2 F1 F1 (+ F1 F2)) F1)

$ (CLOSURE-VARIABLES FIB)                                    ;FIB variables
(F1 F2)
```

6.6 Turtle Graphics

GRAPHICS.LSP provides a set of functions for plotting on the console screen using **turtle graphics** commands similar to those used in the LOGO computer language.

SIN-DEG [*angle*] Function
COS-DEG [*angle*] Function

If *angle* is a number, (SIN-DEG *angle*) returns the sine of *angle* degrees and (COS-DEG *angle*) returns the cosine of *angle* degrees.

```
$ (SIN-DEG 45)
0.7071067

$ (COS-DEG 0)
1
```

DOT [x, y] Function
LINE [x_1, y_1, x_2, y_2] Function

If (x, y) is a valid turtle graphics coordinate, (DOT x y) plots a dot at (x, y) in the color determined by SETCOLOR.

If (x_1, y_1) and (x_2, y_2) are valid turtle graphics coordinates, (LINE x_1 y_1 x_2 y_2) draws a line from (x_1, y_1) to (x_2, y_2) in the color determined by SETCOLOR.

```
$ (DOT 0 0)                                          ;Plot a point
NONE

$ (LINE 0 0 30 50)                                   ;Draw a line
NONE
```

SETPOS [*x*, *y*] Function
SETHEADING [*angle*] Function
SETCOLOR [*color*] Function

If (*x*, *y*) is a valid turtle graphics coordinate, (SETPOS *x y*) positions the turtle at (*x*, *y*). SP can be used as an abbreviation for SETPOS.

If *angle* is a number, (SETHEADING *angle*) points the turtle to a heading of *angle* degrees. SH can be used as an abbreviation for SETHEADING.

If *color* is a valid color number for the current video mode (see Appendix B), (SETCOLOR *color*) sets the color of the turtle's track to *color*. SC can be used as an abbreviation for SETCOLOR.

PENDOWN [ ] Function
PENUP [ ] Function
CLR [ ] Function

(PENDOWN) causes the track of the turtle to be plotted. PD can be used as an abbreviation for PENDOWN.

(PENUP) causes the track to *not* be plotted. PU can be used as an abbreviation for PENUP.

(CLR) clears the screen and restores the turtle to its initial state: pendown, track color white, heading 0 degrees, at position (0, 0).

RIGHT [*angle*] Function
LEFT [*angle*] Function

If *angle* is a number, (RIGHT *angle*) turns the turtle right (clockwise) *angle* degrees and returns the turtle's new heading. If *angle* is a number, (LEFT *angle*) turns the turtle left (counter-clockwise) *angle* degrees and returns the turtle's new heading. If *angle* is negative, the turtle is turned –*angle* degrees in the opposite direction. RT and LT can be used as abbreviations for RIGHT and LEFT respectively.

```
$ (RIGHT 45)                                    ;Turn right 45 degrees
45

$ (LT -45)                                      ;Turn right 45 degrees
90
```

FORWARD [*distance*] Function
BACK [*distance*] Function

If *distance* is a number, (FORWARD *distance*) moves the turtle forward *distance* pixels and returns the symbol NONE. If *distance* is a number, (BACK *distance*) moves the turtle backward *distance* pixels and returns the symbol NONE. If *distance* is negative, the turtle is moved *–distance* pixels in the opposite direction. Note that FORWARD and BACK perform the necessary rounding so that the turtle ends up exactly on a pixel. FD and BK can be used as abbreviations for FORWARD and BACK respectively.

```
$ (FORWARD 50)                                    ;Move forward 50 pixels
NONE

$ (BK -50)                                        ;Move forward 50 pixels
NONE
```

TURTLE [ ] Function
TURTLE-WINDOW [ ] Function

(TURTLE) initializes the turtle parameters to their default values and starts the turtle graphics read-eval-draw loop. The loop first displays the prompt Command: in a four line input window at the bottom of the screen. Next it waits for the user to enter a turtle graphics or standard LISP command. Finally it executes the command in the turtle graphics window at the top of the screen. This loop is repeated until the user enters the command QUIT.

TURTLE-WINDOW is a helper function that creates the appropriate turtle graphics window for the current video mode.

The following sample functions are also defined in the file GRAPHICS.LSP. They provide examples that use the **muLISP** turtle graphics functions.

(POLY *side angle*) draws a polygon whose sides are *side* pixels long and whose turn angle is *angle* degrees.

(CORNER-POLY *side angle*) recursively draws polygons that have similar half-sized polygons at each outside corner until the sides of the polygons are reduced to a single pixel.

(SPIRAL *side angle decrement*) draws a spiral whose initial side is *side* pixels long, whose turn angle is *angle* degrees, and whose sides are decremented *decrement* pixels after each turn until the sides disappear.

(SPIROLAT *side angle decrement*) draws a spirolateral by repeatedly calling SPIRAL until the figure is closed (i.e. the total amount of turning reaches 0 modulo 360).

(C-CURVE *depth*) draws the popular "C" curve to *depth* levels of recursion.

(D-CURVE *depth*) draws the intricate "Dragon" curve to *depth* levels of recursion.

6.7 Mouse Interface

MOUSE.LSP provides a set of functions for reading the position and button status of the **Microsoft Mouse**[tm] or equivalent mouse pointing device from within **muLISP**. Before using these functions the mouse device driver software most be properly installed (see the Mouse Reference Manual for details).

MOUSE-RESET [ ] Function

If the mouse hardware and software have been correctly installed, (MOUSE-RESET) resets the mouse parameters to their default values and returns T, otherwise it returns NIL. The mouse should be reset whenever the video mode is changed (see Section 5.21). The default value of the mouse parameters are:

Parameter	Value
Mouse cursor position	screen center (320, 100)
Mouse cursor horizontal limits	0 to 639
Mouse cursor vertical limits	0 to 199
Mouse cursor shape (graphics)	arrow
Mouse cursor shape (text)	inverse video box
Mouse cursor display byte	off (-1)
Mouse horizontal aspect ratio	8:8
Mouse vertical aspect ratio	16:8
Light pen emulation	enabled
User-defined call mask	all zeros

```
$ (MOUSE-RESET)                                    ;Reset mouse
T
```

SHOW-MOUSE [ ] Function
HIDE-MOUSE [ ] Function

(SHOW-MOUSE) causes the mouse cursor to be displayed on the console screen and returns NIL. (HIDE-MOUSE) causes the mouse cursor not to be displayed and returns NIL. Note that even when the mouse cursor is not displayed, the mouse software tracks the movement of the mouse.

If the **mouse cursor display byte** maintained by the mouse software is zero, the mouse cursor is displayed. If the byte is nonzero, the mouse cursor is not displayed. SHOW-MOUSE and HIDE-MOUSE respectively increment and decrement this byte. Thus three consecutive calls to HIDE-MOUSE will require three calls to SHOW-MOUSE before the mouse cursor is redisplayed.

```
$ (SHOW-MOUSE)                                          ;Shows mouse
NIL

$ (HIDE-MOUSE)                                          ;Hides mouse
NIL
```

MOUSE-POSITION [*x*, *y*] Function
MOUSE-X Variable
MOUSE-Y Variable

If *x* and *y* are non-negative integers less than 640 and 200 respectively, (MOUSE-POSITION *x y*) positions the mouse cursor at the coordinate (*x*, *y*) on the screen and returns NIL. (MOUSE-POSITION) sets the variables *MOUSE-X* and *MOUSE-Y* to the current x and y coordinates of the mouse cursor and returns the y coordinate.

The mouse uses a coordinate system relative to the upper left corner of the full screen (*not* the home position of the current **muLISP** window). Thus the coordinate (0, 0) is the upper left corner of the screen and the coordinate (639, 199) is the lower right corner.

```
$ (MOUSE-POSITION 320 100)                              ;Center mouse on screen
NIL

$ (MOUSE-POSITION)                                      ;y coordinate
100

$ *MOUSE-X*                                             ;x coordinate
320
```

MOUSE-STATUS [ ] Function
LEFT-BUTTON Variable
RIGHT-BUTTON Variable

(MOUSE-STATUS) sets the variable *LEFT-BUTTON* to T if the left mouse button is currently being pressed, otherwise MOUSE-STATUS sets it to NIL. Also MOUSE-STATUS sets *RIGHT-BUTTON* to the status of the right mouse button. Finally, it returns the new value of *RIGHT-BUTTON*.

```
$ (MOUSE-STATUS)                                    ;Check mouse button status
T

$ *LEFT-BUTTON*                                      ;Left button not pressed
NIL

$ *RIGHT-BUTTON*                                     ;Right button pressed
T
```

BUTTON-PRESSES [*button*] Function
BUTTON-RELEASES [*button*] Function

(BUTTON-PRESSES 'LEFT) sets the variables *MOUSE-X* and *MOUSE-Y* to the coordinates of the mouse cursor when the left button was last pressed. It returns the number of times the left mouse button has been pressed since BUTTON-PRESSES was last called on the left button. (BUTTON-PRESSES 'RIGHT) does the same thing for the right mouse button. In either case, BUTTON-PRESSES sets the variables *LEFT-BUTTON* and *RIGHT-BUTTON* to the current button status (see MOUSE-STATUS).

Similarly, BUTTON-RELEASES sets the variables *MOUSE-X* and *MOUSE-Y* to the coordinates of the mouse cursor when the left (right) button was last released. It returns the number of times the left (right) button has been released since BUTTON-RELEASES was last called.

```
$ (BUTTON-PRESSES 'LEFT)                          ;Left button pressed 3 times
3

$ (BUTTON-RELEASES 'RIGHT)                         ;Right button not released
0
```

MOUSE-BOX [x_1, y_1, x_2, y_2] Function

If (x_1, y_1) and (x_2, y_2) are valid mouse coordinates, (MOUSE-BOX x_1 y_1 x_2 y_2) limits the mouse cursor to the rectangular box defined by these coordinates, moves the mouse cursor into the box, and returns NIL.

Note that even if the mouse cursor has been limited to a portion of the screen, the coordinates returned by the function MOUSE-POSITION are still relative to the full screen.

```
$ (MOUSE-BOX 300 90 340 110)                    ;Limit mouse to small box
NIL                                              ;in the center of screen

$ (MOUSE-POSITION)                                        ;Mouse is in upper
90                                                       ;left corner of box

$ *MOUSE-X*
300
```

Chapter 7. Program Development System

muLISP programs can be developed using a standard text editor suitable for your computer and operating system. Program development using such a nonresident editor consists of repeatedly loading the editor, editing the program, saving the program, loading the program into **muLISP**, debugging the program, reloading the editor, etc. Although you can use an editor with which you are familiar, this method of program development is inefficient and cumbersome.

These problems are conveniently resolved by the **muLISP** program development system. Its *resident* editor and debugger greatly reduce program development time by taking advantage of the interactive nature of LISP: programs can be created, tested, and debugged all from within **muLISP**. This encourages the incremental approach to programming and simplifies and shortens the edit-load-debug-reedit cycle.

7.1 Building and Saving the PDS

The editor and debugger are windowing applications that use the services provided by the window manager. The editor is distributed as the file EDIT.LSP. The debugger is distributed as the file DEBUG.LSP. The window manager is distributed as the file WINDOWS.LSP.

If these files are on the drive and directory specified by *pathname*, start **muLISP** (see Section 2.1) and then issue a command of the form

(LOAD '*pathname*\\EDIT)

to load both the editor and the window manager. After loading for several seconds, the prompt

Make editor WordStar-like or EMACS-like? (W/E)

is displayed and the system waits for you to press W to make the editor WordStar-like or to press E to make the editor EMACS-like. If you have no preference, we recommend the WordStar-like option.

Next issue a command of the form

(LOAD '*pathname*\\DEBUG)

to load the debugger. After both the editor and debugger have been loaded and the **muLISP** prompt is displayed, issue a command of the form

(SAVE '*pathname*\\PDS)

to save the program development system as the file PDS.SYS on the drive and directory specified by *pathname*. *pathname* is optional, defaulting to the current drive and directory.

This time, just press the **Esc** key to actually start the editor running. In the future, issue a DOS command of the form

MULISP *pathname*\PDS

to load **muLISP** and the program development system. A DOS command of the form

MULISP *pathname*\PDS *filename*

loads the program development system, and then loads *filename* into the edit buffer.

7.2 The Editor

The **muLISP** editor is a general purpose, screen-oriented source file editor. It continuously displays a "window" or "picture" of the text being edited. Thus, changes made to the text are seen immediately.

The editor has been enhanced to facilitate the editing of LISP source files. It provides commands for editing list structures and for evaluating expressions (see Section 7.2.4.7). The editor makes the balancing of parentheses easy by blinking matching parentheses.

You can make the editor's control key commands either WordStar-like or EMACS-like (see Section 7.1). If desired, you can even modify its control key commands to mimic the behavior of a text editor with which you are already familiar.

The general strategy for using the editor can be summed up in two steps:

- first, move the cursor to where you want to edit;
- second, insert, delete, or change the text at that point.

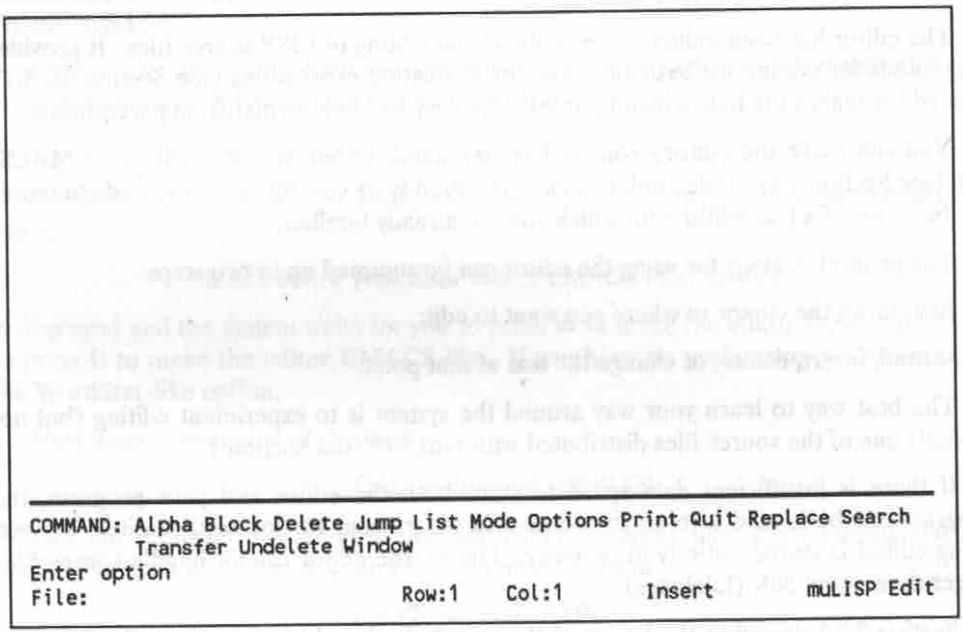
The best way to learn your way around the system is to experiment editing (but *not* saving!) one of the source files distributed with **muLISP** (see Section 1.2).

If there is insufficient data space to store both the editor and your program, the program can be loaded without the editor. In the current version of the editor, the text being edited is stored entirely in memory. Hence, the editor cannot handle source files larger than about 50K (kilobytes).

Section 7.2.1 describes the layout of the screen displayed by the editor. Section 7.2.2 describes how to select options using the command mode menu. Section 7.2.3 summarizes the command mode menu options. Section 7.2.4 describes the control key commands available when the editor is in the edit mode. Section 7.2.5 describes how the editor stores and manages the text displayed on the screen, in case you want to modify the editor.

7.2.1 The Editor Screen

When it starts, the editor clears the screen and displays a screen similar to the following:


COMMAND: Alpha Block Delete Jump List Mode Options Print Quit Replace Search
Transfer Undelete Window
Enter option
File:
Row:1 Col:1 Insert muLISP Edit

The **muLISP** Editor Screen

The bottom line of the editor screen is called the **status line**. It displays

- the file being edited,
- the row and column position of the cursor,
- the text entry mode if in insert mode, and
- the name of the current windowing application.

Immediately above the status line is the **prompt line**. This line displays what you are supposed to do. Initially, it displays Enter option, meaning that you are expected to select an option from the menu of options.

The two lines above the prompt line are the **option lines**. The words following COMMAND are the **menu options**.

The remainder of the screen above the horizonal line is the **work window**. The editor displays the file being edited in this area.

7.2.2 Selecting Menu Options

When the editor is in **command mode**, one of the menu options is highlighted in inverse video and the prompt line displays the prompt Enter option. Each time you press the **Space** bar or the **Tab** key, the highlight moves to the next option. Each time you press **Backspace** or **Shift-Tab**, the highlight moves to the previous option. To select an option, highlight the desired menu option and then press the **Enter** key.

Note that one letter of each option name is capitalized. Normally the first letter is capitalized unless there is another option having the same first letter. Typing the capitalized letter in an option name is a faster way to select the option. It does not matter whether you type the letter in upper or lower case.

Many options invoke sub-menus, which may in turn invoke sub-sub-menus, etc. For brevity, this manual refers to a sequence of options entered from successive menus by concatenating the menu option names. For example, the **Window Open** option is issued by selecting the **Window** top-level menu option and then selecting the **Open** submenu option.

After looking at a sub-menu, you can return to the next higher-level menu by pressing the **Esc** key. Note that **Esc** is also used to quit edit mode and switch to command mode (see Section 7.2.4).

Some menu option commands display one or more **selection fields**, each with its own name, followed by a colon and a list of two or more selections. For example, if you issue an **Options Display** command, it displays

Mode: **Text** Graphics Reso: Medium (High) Text:(Large) Small Set: Std (Extended)
Adapter: MDA (CGA) EGA MCGA VGA Hercules AT&T T3100 95LX PCJr Other

This command contains five selection fields: Mode, Reso(lution), Text, Set, and Adapter. Initially, the current display mode (either Text or Graphics) is highlighted. The highlight indicates that the mode field is the **active field**. The current selections in the other fields are parenthesized. The parentheses indicate that these fields are inactive.

Press the **Tab** key to move the active field right. Press **Shift-Tab** to move the active field left. To make a selection, press **Space** or **Backspace** to highlight the desired selection, or type the first capitalized letter in the selection's name. The latter method of making a selection also moves the active field right. Press **Enter** to invoke your selections. Alternatively, press **Esc** to abort the command without making any changes.

Some menu option commands display one or more **data entry fields**, each with its own name, followed by a colon. If a data entry field is active, a blinking cursor is displayed in the field and you can edit or overwrite the value displayed in the field. For example, if you issue a **Search** command, it displays the prompt

String:

and waits for you to enter a string.

If you press a key for which there is no command, a brief **error beep** is sounded. If you find the beep annoying, use the **Options Mute** command to turn it off.

7.2.3 Command Mode Menu Options

The options described in this subsection can be invoked when the editor is in command mode. Many of these operations can also be accomplished more quickly using control keys when the editor is in edit mode (see Section 7.2.4). The following is an alphabetical summary of the command mode menu options:

Alpha switches the editor from command mode to edit mode (see Section 7.2.4).

Block Begin places the beginning block marker at the current cursor position.

Block Copy copies the text delimited by the beginning and ending block markers and puts the copy at the current cursor position.

Block Delete deletes the text delimited by the beginning and ending block markers.

Block End places the ending block marker at the current cursor position.

Block Jump Begin moves the cursor to the beginning block marker.

Block Jump End moves the cursor to the ending block marker.

Block Move moves the text delimited by the beginning and ending block markers to the current cursor position.

Block Write writes a copy of the text delimited by the beginning and ending block markers to the file you enter following the prompt BLOCK WRITE file. If the file already exists, the warning query Overwrite existing file (Y/N)? is made to confirm the decision.

Delete Block deletes the text delimited by the beginning and ending block markers.

Delete Character deletes the character under the cursor.

Delete Line deletes the line the cursor is on, including the new line character at the right end of the line.

Delete Right-end deletes the right end of the current line from the cursor to the end of the line but *not* the new line character.

Delete Word deletes the word under and to the right of the cursor.

Jump Bottom moves the cursor to the bottom row of the text window.

Jump End moves the text window and the cursor to the end of the file.

Jump Last moves the cursor to the position it had before the last major cursor movement command.

Jump Start moves the text window and the cursor to the beginning of the file.

Jump Top moves the cursor to the top row of the text window.

List Back moves the cursor backward one list.

List Delete deletes the list under and to the right of the cursor.

List Eval evaluates the expression under and to the right of the cursor. This feature makes it possible to selectively redefine individual function definitions in a file so they can be tested without having to save the entire file on disk and then load it back into **muLISP**. If there is an open debug window on the screen (see Section 7.3), the result of the expression evaluation is displayed in that window.

List Forward moves the cursor forward one list.

List Next moves the cursor down to the beginning of the line whose first character is neither a blank nor a semicolon (i.e. usually the beginning of the next definition).

Mode displays the current text entry mode and lets you select the new mode. In **insert mode**, characters under and to the right of the cursor are moved right to make room for a new character. In **overwrite mode**, the character under the cursor is replaced by a new character.

Options Color Menu displays the current menu colors and lets you select the new colors. The menu colors include the foreground and background colors of the border lines between windows, the option lines, the prompt line, and the status line.

Options Color Work displays the current work window colors and lets you select the new colors. The work window colors include the foreground and background colors of the text being edited.

Options Display displays the current display mode and lets you select the new mode. The display mode includes the screen mode (Text or Graphics), the resolution (Medium or High), and the display adapter (MDA, CGA, EGA, MCGA, VGA, Hercules, AT&T, T3100, or PCjr). If the EGA adapter is selected, you are also asked to select a monitor type (Color, Enhanced, or Monochrome).

Options Execute executes the DOS command you enter following the prompt Enter DOS command. If you just press the **Enter** key without entering a command, **muLISP** is temporarily suspended and control passes to the DOS command processor. To return to **muLISP**, type EXIT following the DOS prompt.

Options Mute lets you turn on or off the sounding of an error beep when an invalid key is pressed.

Print Printer prints the contents of the file you enter following the prompt PRINT PRINTER file. Note that if changes are made to text in the edit buffer, the buffer must be saved *before* the file is printed so the changes are reflected in the printout.

Print Layout displays the current printer page layout and lets you enter changes. The page layout includes the length, width, and margins of the print page. Initially the

page length is 66 lines, the page width is 80 characters, the top and bottom margins are 0 lines each, the left margin is 8 characters, and the right margin is 3 characters.

Print Options lets you enter the beginning and ending page numbers to print, control the truncation or wrapping of long lines, and select the output device (printer, console, or both).

Quit abandons the current text, terminates **muLISP**, and returns control to DOS. If changes have been made to the text since it was last saved, the warning query Abandon edited text (Y/N)? is made to confirm the decision.

Replace replaces the first occurrence of the string you enter following the prompt Enter string to replace with the string you enter following the prompt Enter replacement string. The warning query Replace string (Y/N)? is made to confirm the decision. If the string to replace is not found, the cursor is moved to the end of the text in the edit buffer.

Search moves the cursor forward to the first occurrence of the string you enter following the prompt Enter string to find. If the string is not found, the cursor is moved to the end of the text in the edit buffer. This is a case sensitive search.

Transfer Clear abandons the text in the edit buffer and clears the text window screen. If changes have been made to the text since it was last saved, the warning query Abandon edited text (Y/N)? is made to confirm the decision before abandoning the text.

Transfer Delete deletes from the DOS directory the file you enter following the prompt TRANSFER DELETE file.

Transfer Load abandons any existing text in the edit buffer and loads the file you enter following the prompt TRANSFER LOAD file. If changes have been made to the existing text since it was last saved, the warning query Abandon edited text (Y/N)? is made to confirm the decision.

Transfer Merge inserts at the current cursor position the text of the file you enter following the prompt TRANSFER MERGE file.

Transfer Save writes the text in the edit buffer to the file you enter following the prompt TRANSFER SAVE file. The text in the edit buffer remains unchanged. Used periodically during an extended editing session, this command can prevent the catastrophic loss of edits made to a file in the event of a power failure or software error.

Undelete inserts at the current cursor position the text deleted by the most recent multi-character delete command. Single character deletions cannot be undeleted. The **Undelete** command is useful for restoring text that has been mistakenly deleted. Used in conjunction with a delete command, it is also a convenient way to move or copy a word or line of text.

Window Close closes the window you name following the prompt WINDOW CLOSE Window.

Window Designate designates the current window to be a window of the type you select following the prompt WINDOW DESIGNATE Type.

Window Flip flips to the next layer of the current window. Pressing the **F2** function key is an alternative way to flip to the next layer.

Window Goto goes to the window you enter following the prompt WINDOW GOTO Window.

Window Next goes to the next window. Pressing the **F1** function key is an alternative way to go to the next window.

Window Open opens a new layer of the current window and designates it to be a window of the type you select following the prompt WINDOW OPEN Type.

Window Previous goes to the previous window. **Shift-F1** or **Ctrl-F1** are alternative ways to go to the previous window.

Window Split Horizontal splits the current window at the line you enter following the prompt WINDOW SPLIT HORIZONTAL At line.

Window Split Vertical splits the current window at the column you enter following the prompt WINDOW SPLIT VERTICAL At column.

7.2.4 Edit Mode Commands

Use an **Alpha** command to switch the editor from command mode to edit mode (see Section 7.2.3). When the editor is in edit mode, the prompt line displays

Edit text or press ESC for options

and the cursor moves to the text in the work window, positioned at the current text entry point.

As the prompt line indicates, press the **Esc** key to quit edit mode and switch to command mode options (see Section 7.2.3).

To aid in balancing parentheses, if there is a right parenthesis under or to the left of the cursor, the matching left parenthesis will blink. Similarly, if there is a left parenthesis under or to the right of the cursor, the matching right parenthesis will blink.

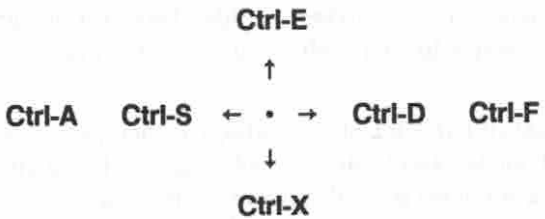
Control keys are used to issue edit mode commands. There is a control key associated with each letter of the alphabet. To enter a control key, press its associated letter while holding down the **Ctrl** key. On PC compatible computers, the arrow keys, usually located at the right end of the keyboard, can also be used to issue edit mode commands.

The **muLISP** editor can be configured to be either WordStar-like or EMACS-like (see Section 7.1). The WordStar-like edit mode commands are summarized in Appendix F. The EMACS-like edit mode commands are summarized in Appendix G. To simplify the following discussion, WordStar-like mode is assumed.

7.2.4.1 Cursor Control Commands

The editor uses a blinking **cursor** to show where the next character you type will appear in the text. The cursor control commands allow you to move the cursor within the text.

In WordStar-like mode the basic cursor control commands are issued using control keys entered using the left hand. The control commands are chosen so that the position of the control key corresponds to the direction the cursor is to move, as illustrated by the following diagram:



These control keys move the cursor up, down, left or right from the dot in the center of the diagram. The best way to get the "feel" of these commands is to experiment with the **muLISP** editor on an existing source file.

Less frequently used edit mode commands are initiated by entering two-character commands. Two-character commands that begin with **Ctrl-Q** usually "amplify" or "reverse" the effect that the second character would have by itself.

The following is a summary of the cursor control commands:

Ctrl-D and **Right arrow** move the cursor forward one character. If the cursor is at the right end of a line, it is moved down to the beginning of the next line.

Ctrl-S and **Left arrow** move the cursor backward one character. If the cursor is at the beginning of a line, it is moved up to the right end of the previous line.

Ctrl-F and **Ctrl-Right arrow** move the cursor forward one word, leaving the cursor at the beginning of a word. A **word** is a string of characters in the text delimited by any non-alphanumeric character or by the beginning or end of a line.

Ctrl-A and **Ctrl-Left arrow** move the cursor backward one word, leaving the cursor at the beginning of a word.

Ctrl-X and **Down arrow** move the cursor down one line.

Ctrl-E and **Up arrow** move the cursor up one line.

Ctrl-Q D and **End** move the cursor to the right end of the current line.

Ctrl-Q S and **Home** move the cursor to the beginning of the current line.

Ctrl-Q X and **Ctrl-End** move the cursor to the bottom of the screen.

Ctrl-Q E and **Ctrl-Home** move the cursor to the top row of the screen.

Ctrl-I and **Tab** move the cursor right to the next **tab stop** (i.e. the next cursor column that is a multiple of 8) when the editor is in overwrite mode. See Section 7.2.4.3 for their effect when the editor is in the insert mode.

Ctrl-Q I and **Shift-Tab** moves the cursor left to the previous tab stop.

Ctrl-M and **Enter** move the cursor down to the beginning of the next line when the editor is in overwrite mode. See Section 7.2.4.3 for their effect when the editor is in insert mode.

Ctrl-J and **Ctrl-Enter** move the cursor to the beginning of the text on the line below the cursor if there is text on the line below the cursor. Otherwise, they move the cursor down a line and automatically indent the cursor based on the number of opening and closing parentheses on the original cursor line.

Ctrl-Q P moves the cursor to the position it had before the last major cursor movement command (e.g. commands such as **Ctrl-Q X** or **Ctrl-Q E**).

Ctrl-K 0, **Ctrl-K 1**, **Ctrl-K 2**, and **Ctrl-K 3** place a correspondingly numbered marker at the current cursor position.

Ctrl-Q 0, **Ctrl-Q 1**, **Ctrl-Q 2**, and **Ctrl-Q 3** move the cursor to the correspondingly numbered marker.

7.2.4.2 Display Control Commands

In general, only part of the text in a file can be shown on the screen at one time. The screen should be thought of as a "window" that displays only a portion of the text. When the editor receives a command that moves the cursor to a position in the text that is not displayed in the window, the text is scrolled up (toward the end of the file) or down (toward the beginning of the file) so the cursor will be displayed.

Ctrl-Z scrolls the text up a line so a new line of text will be displayed at the bottom of the screen. Note that this is equivalent to moving the window down a line. The cursor remains at the same position in the text, moving up with the text. However, if the cursor reaches the top of the screen, it remains there. The cursor never leaves the screen.

Ctrl-W scrolls the text down a line so a new line of text will be displayed at the top of the screen. Note that this is equivalent to moving the window up a line. The cursor remains at the same position in the text, moving down with the text. However, if the cursor reaches the bottom of the screen, it remains there. The cursor never leaves the screen.

Ctrl-C and **PgDn** move the window down (i.e. move the text up) about 3/4 of a full screen, leaving the cursor at the same position on the screen but moved in the text.

Ctrl-R and **PgUp** move the window up (i.e. move the text down) about 3/4 of a full screen, leaving the cursor at the same position on the screen but moved in the text.

Ctrl-Q C and **Ctrl-PgDn** move the window and the cursor to the end of the text.

Ctrl-Q R and **Ctrl-PgUp** move the window and the cursor to the beginning of the text.

7.2.4.3 Entering Text Commands

The editor is always in one of two text entry modes. When in **overwrite mode**, characters and spaces under and to the right of the cursor will be overwritten by new characters as they are entered. This is the easiest way to enter text initially. A mistake can be corrected by simply typing over it with the correction, and text can be "erased" by simply typing over it with spaces.

When the editor is in **insert mode**, characters and spaces under and to the right of the cursor are pushed over to the right to make room for the new characters inserted in front of them. In this mode, the word Insert is displayed on the status line. On PC compatible computers, the height of the cursor is also doubled in this mode.

Ctrl-V and **Ins** switch between insert and overwrite modes. Initially, the editor is in insert mode.

Ctrl-I and **Tab** insert enough spaces in the text to move the cursor right to the next tab stop when the editor is in insert mode. See Section 7.2.4.1 for their effect when the editor is in the overwrite mode.

Ctrl-M and **Enter** insert a new line in the text and move the cursor down to the beginning of the new line when the editor is in insert mode. See Section 7.2.4.1 for their effect when the editor is in overwrite mode.

Ctrl-N inserts a new line in the text in both text entry modes. However, unlike **Enter**, the cursor is *not* moved and remains at the right end of the newly broken line.

Ctrl-P causes the editor to ignore the normal cursor control features of the next character entered, so it can be added to the text like other characters.

7.2.4.4 Deleting Text Commands

In the overwrite mode the simplest way to delete text is to simply type spaces or new characters over the text to be deleted. The explicit text deletion commands in this subsection work in both overwrite and insert modes. The block and list structure commands also provide methods for deleting text (see Sections 7.2.4.5 and 7.2.4.7).

Ctrl-G and **Del** delete the character under the cursor.

Ctrl-H, **Rub**, and **Backspace** delete the character to the immediate left of the cursor and move the cursor backward one character.

Ctrl-T deletes the word under and to the right of the cursor.

Ctrl-Q T deletes the word to the left of the cursor.

Ctrl-Y deletes the line the cursor is on, including the new line character at the end of the line.

Ctrl-Q Y deletes the right end of the current line from the cursor to the end of the line but *not* the new line character.

Ctrl-Q H, **Ctrl-Q Rub**, and **Ctrl-Q Backspace** delete the left end of the line from the cursor to the beginning of the line.

Ctrl-U restores at the current cursor position the text deleted by the most recent multi-character text deletion command. **Ctrl-U** makes it possible to restore text deleted by mistake and also makes it easy to move or copy a line of text.

To *move* a line, place the cursor on the line to be moved, press **Ctrl-Y** to delete the line, move the cursor to the new location for the line, and press **Ctrl-U** to undelete the line. To *copy* a line, press **Ctrl-U** immediately after deleting the line and again after moving the cursor to the new location.

7.2.4.5 Block Commands

The block commands permit a block of text to be moved, copied, deleted, read from a file, or written to a file. Two- and three-character commands that begin with **Ctrl-O** initiate the block commands. The second and third character of **Ctrl-O** commands are mnemonic. Alternatively and in accordance with WordStar, two-character commands that begin with **Ctrl-K** can be used to initiate block commands. But the second character of these **Ctrl-K** commands are not all mnemonic.

Ctrl-O B and **Ctrl-K B** place the beginning block marker at the current cursor position.

Ctrl-O E and **Ctrl-K K** place the ending block marker at the current cursor position.

Ctrl-O M and **Ctrl-K V** move the text delimited by the beginning and ending block markers from its present position to the current cursor position.

Ctrl-O C and **Ctrl-K C** copy the text delimited by the beginning and ending block markers and puts the copy at the current cursor position.

Ctrl-O D and **Ctrl-K Y** delete the text delimited by the beginning and ending block markers.

Ctrl-O W and **Ctrl-K W** write a copy of the text delimited by the beginning and ending block markers to the file you enter following the prompt BLOCK WRITE file. If the file already exists, the warning query Overwrite existing file (Y/N)? is made to confirm the decision.

Ctrl-O J B and **Ctrl-Q B** move the cursor to the beginning block marker.

Ctrl-O J E and **Ctrl-Q K** move the cursor to the ending block marker.

7.2.4.6 Search and Replace Commands

Ctrl-Q F moves the cursor forward to the first occurrence of the string you enter following the prompt Enter string to find. If the string is not found, the cursor is moved to the end of the text in the edit buffer. This is a case sensitive search.

Ctrl-Q A replaces the first occurrence of the string you enter following the prompt Enter string to replace with the string you enter following the prompt Enter replacement string. The warning query Replace string (Y/N)? is made to confirm the decision. If the string to replace is not found, the cursor is moved to the end of the text in the edit buffer.

Ctrl-L repeats the last search or replace command looking for the *next* occurrence of the string. If repeating a replace command, the warning query Replace string (Y/N)? is made to confirm the decision.

Ctrl-Q L repeats the last search and replace command looking for *all* occurrences of the string from the current cursor position to the end of the text. The warning query

Replace string (Y/N)? is made to confirm each decision. Press the **Esc** key to terminate the multiple replacement command.

Ctrl-Q V moves the cursor to the point at which the last search or replace operation was initiated.

7.2.4.7 List Structure Commands

The list structure commands are for editing lists in the text. Two-character commands that begin with **Ctrl-^** initiate the list structure commands. On PC compatible computers, the **Alt** key can also be used to initiate these commands.

Ctrl-^ D and **Alt-D** move the cursor forward one list.

Ctrl-^ S and **Alt-S** move the cursor backward one list.

Ctrl-^ C and **Alt-C** move the cursor down to the beginning of the line whose first character is neither a blank nor a semicolon (i.e. usually the beginning of the next definition).

Ctrl-^ R and **Alt-R** move the cursor up to the beginning of the line whose first character is neither a blank nor a semicolon.

Ctrl-^ T and **Alt-T** delete the list under and to the right of the cursor.

Ctrl-^ I and **Alt-I** evaluate the list under and to the right of the cursor. This feature makes it possible to selectively redefine edited function definitions so they can be tested.

7.2.4.8 File Saving Commands

The file saving commands are used to save the text as a file or abandon the text. These commands are initiated by two-character commands beginning with **Ctrl-K**.

Ctrl-K D and **Ctrl-K S** write the text in the edit buffer to the file you enter following the prompt TRANSFER SAVE file. The text in the edit buffer remains unchanged. Used periodically during an extended editing session, these commands can prevent the catastrophic loss of edits made to a file in the event of a power failure or software error.

Ctrl-K A and **Ctrl-K Q** abandon the text in the edit buffer and clear the text window screen. If changes have been made to the text since it was last saved, the warning query Abandon edited text (Y/N)? is made to confirm the decision before abandoning the text.

Ctrl-K R inserts at the current cursor position the text of the file you enter following the prompt TRANSFER MERGE file.

7.2.5 Theory of Operation

Since the editor is written entirely in **muLISP**, you can modify or even add features to the editor to suit your taste. This subsection briefly describes how text is stored in the edit buffer and how control key commands are processed. If you intend to make changes, we recommend printing a listing of EDIT.LSP.

One of the simplest changes you can make to the editor is to reassign the control keys so the **muLISP** editor will "feel" like an editor with which you are already familiar. When a control key is pressed, the associated function on the association list (a-list) bound to the global variable *EDIT-DEMONS* is called to perform the desired operation. Near the end of EDIT.LSP, *EDIT-DEMONS* is assigned to either the WordStar-like a-list or the EMACS-like a-list based on your choice. You can change the control key commands simply by reassigning *EDIT-DEMONS* to your own a-list.

More ambitious changes, such as modifying or adding new edit mode commands to the editor, require a thorough understanding of how the text is stored and how to use the editor's existing text primitives to manipulate the text and change the position of the point. Throughout this discussion, the **point** is the position in the text between the character the cursor is on and the character to its immediate left.

The text is stored as a list of symbols whose print names are the string of characters for each line of text. *BELOW-POINT* is the list for the text below (after) the point. *ABOVE-POINT* is the list for the text above (before) the point. The first line below and above the point are the first elements of *BELOW-POINT* and *ABOVE-POINT* respectively. The second line is the second element, etc.

POINT-ROW and *POINT-COL* are non-negative integers that store the current row and column of the point relative to the start of the text. *CURSOR-ROW* and *CURSOR-COL* store the current row and column of the point relative to the top left corner of the current edit window.

The function JUMP-TO-POSN should be used to move the point within the text since it automatically updates the above variables and scrolls the screen if necessary. (JUMP-TO-POSN *row column*) moves the point to text row *row* and text column *column*.

Editor Control Variables

This table summarizes the text control variables.

Variable	Description
ABOVE-POINT	a list of the text above the point
BELOW-POINT	a list of the text below the point
CURSOR-COL	cursor column position in the edit window
CURSOR-ROW	cursor row position in the edit window
DELETED-TEXT	most recently deleted text
EDIT-MODE	T if in edit mode, NIL if in command mode
END-COL	ending column of the block in the text
END-ROW	ending row of the block in the text
FILE-NAME	file currently being edited
FIND-COL	text column of the last search and replace
FIND-ROW	text row of the last search and replace
FIND-STRG	current "Find string"
INSERT-MODE	T if in the insert mode, NIL if in overwrite mode
LAST-COL	text column of the last significant move
LAST-ROW	text row of the last significant move
POINT-COL	text column of the point
POINT-ROW	text row of the point
REPL-STRG	current "Replace string"
SCREEN	list of print names currently being displayed
START-COL	starting column of the block in the text
START-ROW	starting row of the block in the text
STAT-COL	column number displayed in status window
STAT-ROW	row number being displayed in status window
TEXT-DIRTY	nonNIL if the text has been changed since it was last saved

This table summarizes the printer control variables and their initial value. Most of these variables can be changed using a **Print Layout** command. The value of the rest can be changed in EDIT.LSP as desired.

Constant	Value	Description
PAGE-LINES	66	Page length in lines
PAGE-COLUMNS	80	Page width in characters
TOP-MARGIN	0	Top margin in lines
BOTTOM-MARGIN	0	Bottom margin in lines
LEFT-MARGIN	8	Left margin in characters
RIGHT-MARGIN	3	Right margin in characters
FOOTER-MARGIN	2	Footer margin in lines
INIT-PRINTER	""	Printer's initialization string
STOP-PRINTER	""	Printer's termination string
PRINTER-DEVICE	PRN	Printer device name
USE-FORMFEED	NIL	Output FORMFEEDs between pages if nonNIL

This table summarizes some miscellaneous control variables and their initial value. Their value can be changed in EDIT.LSP as desired.

Constant	Value	Description
DEFAULT-TYPE	LSP	Default file name extension
CASE-IGNORE	NIL	Ignore case during string searches if nonNIL
USE-TABS	T	Use TABs when saving files if nonNIL

7.3 The Debugger

The **muLISP** debugger is designed to help you test and debug programs. It allows you to trace functions so you can observe the progress of a program and/or stop a program at critical points so you can examine and/or modify its control variables. The debugger also allows you to have the trace output displayed in one window and program output displayed in another window.

To start the debugger, load the program development system (see Section 7.1). When the system starts, it begins with an edit window (see Section 7.2). Use a **Window Open** or **Window Split** command to create a debug window.

When a debug window is active, muLISP Debug is displayed at the right end of the status line. The percentage of *free* space in the data and code spaces at the time of the last garbage collection is also displayed on the status line.

Section 7.3.1 summarizes the debugger's command mode menu options. Section 7.3.2 shows typical trace output.

7.3.1 Command Mode Menu Options

The options described in this subsection are selected in the same way as the editor's menu options (see Section 7.2.2). The following is an alphabetical summary of the command mode menu options:

Author allows you to enter forms for evaluation just as you do following the **muLISP** dollar-sign prompt. The input line and result are displayed in the debug window.

Backtrace displays the current nesting of traced function calls. This option is only available during a Step mode break (see Section 7.3.2).

Delete deletes from the DOS directory the file named by you following the prompt DELETE file.

History displays the most recent *HISTORY-LENGTH* lines of trace history even though the trace mode may have been Off during the trace. Using the **Author** option, you can change the positive integer value of *HISTORY-LENGTH* as desired. Its initial value is 25.

Load loads the source file named by you following the prompt LOAD file. As each expression in the file is read and evaluated, its value is displayed in the debug window if the control variable *SHOW-LOAD* is nonNIL.

Mode lets you set the trace mode to Continuous, Step, or Off as selected by you following the prompt MODE. Section 7.3.2 describes the effect the trace mode has on function tracing. This command also lets you direct future trace output to a printer in order to produce a "hard copy" record of a long trace.

Options Color Menu displays the current menu colors and lets you select the new colors. The menu colors include the foreground and background colors of the border lines between windows, the option lines, the prompt line, and the status line.

Options Color Work displays the current work window colors and lets you select the new colors. The work window colors include the foreground and background colors of the work window.

Options Display displays the current display mode and lets you select the new mode. The display mode includes the screen mode (Text or Graphics), the resolution (Medium or High), and the display adapter (MDA, CGA, EGA, MCGA, VGA, Hercules, AT&T, T3100, or PCjr). If the EGA adapter is selected, you are also asked to select a monitor type (Color, Enhanced, or Monochrome).

Options Execute executes the DOS command you enter following the prompt Enter DOS command. If you just press **Enter** without entering a command, **muLISP** is temporarily suspended and control passes to the DOS command processor. To return to **muLISP**, type EXIT following the DOS prompt.

Options Mute lets you turn on or off the sounding of an error beep when an invalid key is pressed.

Print Printer prints the contents of the file you enter following the prompt PRINT file.

Print Layout displays the current printer page layout and lets you enter changes. The page layout includes the length, width, and margins of the print page. Initially the page length is 66 lines, the page width is 80 characters, the top and bottom margins are 0 lines each, the left margin is 8 characters, and the right margin is 3 characters.

Print Options lets you enter the beginning and ending page numbers to print.

Quit terminates **muLISP** and returns control to DOS. This option is *not* available during a Step mode break.

Resume proceeds from a Step mode break. This option is only available during a Step mode break.

Trace traces the function entered by you following the prompt TRACE function. If you enter the name of a primitively defined function, the system also prompts you to enter the name of the user-defined function in which the primitive function is to be traced. Only the special forms SETQ, INCQ, and DECQ can be traced. When the traced function is called by any function, its arguments and returned value are displayed in the debug window (see Section 7.3.2).

traceIn allows you to trace *only* calls to a function that are made from a specified user-defined function. You are prompted for the function to be traced and then for the user-defined function which calls the traced function. Primitively defined functions, user-defined functions, and the special forms SETQ, INCQ, and DECQ can be traced in user-defined functions. When the traced function is called by the function in which

it is traced, its arguments and returned value are displayed in the debug window (see Section 7.3.2).

Untrace All untraces all the functions in the current debug window.

Untrace One untraces the function entered by you following the prompt UNTRACE function. If you enter the name of a function that has been traced in one or more functions, the system also prompts you to enter the name of the user-defined function in which the primitive function is to be untraced.

Window performs the same windowing operations as the editor's window command mode options (see Section 7.2.3). These options are *not* available during a Step mode break.

7.3.2 Debugger Display

Throughout this discussion of the debugger, we assume that the functions APP and REV have been defined as follows:

```
(DEFUN APP (LST1 LST2)
  ((NULL LST1) LST2)
  (CONS (CAR LST1) (APP (CDR LST1) LST2)) )

(DEFUN REV (LST)
  ((NULL LST) NIL)
  (APP (REV (CDR LST)) (LIST (CAR LST))) )
```

APP and REV are functionally equivalent to the primitively defined functions APPEND and REVERSE. Since APP and REV are less general and less efficient than their machine language counterparts, we have given them different names.

To display a continuous trace of function calls and returns, set the trace mode to Continuous. When a traced function is called, the function's name is displayed, followed by a list of its arguments. When a traced function returns, its name, an equal sign, and its value are displayed.

For example, use the **Trace** option to trace REV and APP. Then use the **Author** option to issue the following call on REV:

```
(REV '(A B))
```

This will produce the following trace in the work window:

```
REV [(A B)]
  REV [(B)]
    REV [NIL]
    REV = NIL
    APP [NIL, (B)]
    APP = (B)
  REV = (B)
  APP [(B), (A)]
    APP [NIL, (A)]
    APP = (A)
  APP = (B A)
REV = (B A)
```

Note how indentation is used to graphically indicate the nesting of function calls. If trace output is scrolling too rapidly to read, use **Ctrl-S** to temporarily freeze and unfreeze the display.

To avoid producing prodigious amounts of useless trace output, use the **Mode** option to set the trace mode to Off.

Even if the trace mode is Off, you can use the **History** option to display the most recent trace output that had been suppressed. A fixed length buffer is used to store the history. The buffer length is set by the control variable *HISTORY-LENGTH*. Its initial value is 25. It can be changed using the **Author** option.

To suspend evaluation of a program when a traced function is called, use the **Mode** option to set the trace mode to Step mode. When evaluation is suspended, the debug window menu appears with **Resume** as an additional option. During the suspension, you can

- use the **Author** option to examine and/or change the value of program variables,
- use the **Backtrace** option to display a backtrace of the current nesting of traced function calls,
- use the **History** option to display the most recent trace history, and
- use the **Mode** option to switch the trace mode to Continuous or Off, thereby preventing further suspensions.

Select the **Resume** option when you are ready to continue evaluation of the program.

7.4 The Window Manager

The **muLISP** window manager provides a flexible, menu-driven interface for window-oriented application programs. The many windowing services the manager furnishes greatly simplify the writing of such programs. It presents the end-user with a standard interface that is easy to learn and use.

The editor and debugger are programs that are built on the window manager. This section describes how to write your own windowing applications.

ADDRESS.LSP is a simple data entry program that illustrates the basic concepts involved in writing a windowing application. We recommend printing a listing of ADDRESS.LSP since we will refer to it throughout the remainder of this section.

The window manager is distributed as the file WINDOWS.LSP. Normally windowing applications automatically chain load the window manager so it is unnecessary to explicitly load WINDOWS.LSP (see Section 7.1). Code at the end of ADDRESS.LSP illustrates how to chain load the window manager.

The entry point for the window manager is the function WINDOWS. If you want the window manager to start automatically when it is saved as a SYS, COM, or EXE file, assign DRIVER the value WINDOWS (see Section 5.10.1).

There can be several windowing applications loaded at one time. The initial window displayed by the window manager is the one assigned to the control variable *INIT-WINDOW*. If you intend for your program to be the initial application, set *INIT-WINDOW* to the name of its top-level entry function. For example, the entry function for the data entry program is "Address".

The control variable *WINDOW-TYPES* is a list of the top-level entry functions for each of the windowing applications currently loaded. You should merge the name of your program's entry function with the existing list of window types. When a new window is being opened, these names are presented to the user so one can be selected. For this reason the name of your program's top-level entry function should be a capitalized string naming the application.

7.4.1 Window Manager Commands

The top-level entry function of an application is called by the window manager when an instance of the application is needed to perform some operation. The command to perform the operation is passed as the first argument to the entry function. The remaining arguments contain the state information about the instance of the application. The entry function must be able to handle the following commands from the window manager:

CREATE-WINDOW This command directs the entry function to return a list of the state information appropriate for a newly created window. The first element of the list is the application's name. The remaining elements correspond to the values of all but the first argument of the top-level entry function. This command should *not* send output to the screen.

If the window manager is started automatically when it is loaded as a SYS, COM, or EXE file, the global variable *COMMAND-LINE* is set to the DOS command line string. In the code for the CREATE-WINDOW command, the value of *COMMAND-LINE* can be used as the initial file name to process. See ADDRESS.LSP as an example.

CLOSE-WINDOW This command is issued to the entry function when the window manager desires to close an instance of an application. The entry function should return NIL if it is not acceptable to close the window. In the data entry program this is done by the predicate ABANDON-RECORDS which asks the user if it is all right to abandon records that have been changed. This command should use the prompt line to request information from the user and should *not* otherwise send output to the screen.

UPDATE-WINDOW This command directs the entry function to clear and then refresh the display in the working window of an instance of an application. In the data entry program this is done by the function UPDATE-ADDRESS which displays the data entry prompts in dim characters and the current record in bold.

RUN-WINDOW This command directs the entry function to actually run an instance of an application. For the data entry program this involves displaying information on the status line using the function ADDRESS-STATUS. Then the program enters a loop which calls upon DISPLAY-RECORD to display the current record and then calls upon EXECUTE-OPTION to execute an option selected by the user.

7.4.2 Window Manager Utilities

The window manager provides several useful utility functions that simplify the writing of windowing applications and present the user with a consistent interface. You can edit the file WINDOWS.LSP to see the definitions of the utility functions described in this subsection.

CHANGE-WINDOW [ ] Function

CHANGE-WINDOW provides the standard submenu for the top-level **Window** option. Facilities are provided for opening, closing, splitting, and changing windows.

CONTINUE-PROMPT [ ] Function

CONTINUE-PROMPT displays Press any key to continue on the prompt line, clears the console input buffer, and waits for the user to press any key other than **Ctrl-S**. It is useful for giving the user a chance to read information displayed on the screen before it is erased.

ERROR-BEEP [ ] Function

If warning tones are not being muted, ERROR-BEEP clears the console input buffer and sounds a short error beep. It is useful for warning the user of an invalid input or some other error condition.

EXECUTE-OPTION [*prompt*, *option-tree*] Function

EXECUTE-OPTION provides a consistent menu-driven interface for programs. Its first argument is the desired option prompt message. Its second argument is the application program's option tree. An entry in the option tree consists of a capitalized option name dotted with the function to execute when the option is selected. See the assignment to *ADDRESS-OPTIONS* in ADDRESS.LSP for an example of an option tree.

PROMPT-YN [*prompt*] Function

PROMPT-YN displays *prompt* on prompt line and waits for the user to select Y or N. If Y is selected, PROMPT-YN returns T, otherwise it returns NIL.

QUIT-PROGRAM [ ] Function

QUIT-PROGRAM requests the window manager to terminate all running applications and return control to DOS. Before doing so, the window manager ensures that all running applications approve of the termination. If some application (perhaps the application that issued the QUIT-PROGRAM request) disapproves, the window manager uses a RUN-WINDOW command to return control to the application that issued the QUIT-PROGRAM request. Normally QUIT-PROGRAM is called when a **Quit** option is chosen.

SET-MENU-COLOR [ ] Function
SET-WORK-COLOR [ ] Function
SET-DISPLAY [ ] Function
SET-INSERT [ ] Function
SET-MUTE [ ] Function
GO-DOS [ ] Function

These functions provide a range of services that are often included in the **Options** submenu (see Section 7.2.3).

SET-MENU-COLOR displays the current menu colors and lets the user select the new colors (see the **Options Color Menu** option).

SET-WORK-COLOR displays the current work window colors and lets the user select the new colors (see the **Options Color Work** option).

SET-DISPLAY displays the current video display mode and lets the user select the new mode (see the **Options Display** option).

SET-INSERT lets the user select either the insert or overwrite text entry mode (see the **Mode** option).

SET-MUTE lets the user mute warning messages (see the **Options Mute** option).

GO-DOS lets the user enter a DOS command (see the **Options Execute** option).

SHOW-PROMPT [*prompt*] Function

SHOW-PROMPT displays *prompt* on the prompt line and updates the variable *PROMPT* which stores what is currently being displayed in the prompt line.

UPDATE-STATE [*arglist*] Function

If *arglist* is a list of an application's state variables, (UPDATE-STATE *arglist*) updates the values of the variables in the global variable *WINDOWS*. UPDATE-STATE should be the last function called by the top-level entry function if there is any possibility that the state has changed since the entry function was called by the window manager. In ADDRESS.LSP the call to UPDATE-STATE is enclosed in an UNWIND-PROTECT form after the RUN-WINDOW loop. The call is protected in this way since the loop is exited by means of a throw.

WINDOWS [*command-line*] Function

WINDOWS is the entry point for the window manager. If you want the window manager to start automatically when it is saved as a SYS, COM, or EXE file, assign DRIVER the value WINDOWS (see Section 5.10.1).

After performing some initialization and displaying the borders for a single window, WINDOWS runs the initial program, *INIT-WINDOW*. When an application terminates by means of a throw, the value of *THROW-TAG* is used to determine the action that WINDOWS takes.

Chapter 8. Compiler

The incremental **muLISP** compiler can greatly improve the execution speed of programs by transforming their function definitions into native machine code. With the exception of faster execution, the behavior of a function does not change after it is compiled: it can be called at the read-eval-print loop level, it can be used in the definitions of other functions, it can be applied indirectly through APPLY, FUNCALL, or the mapping functions, and so forth.

After compiling the functions in a program, the function DELETE-COMPILER can be used to regain most of the memory space used by the compiler. Finally, the function SAVE can be used to save the program, including the compiled machine code, as a fast loading SYS file or COM file (see Section 2.6 and Section 5.9.4).

The compiler is fully incremental in the sense that compiled functions call the most recent definition of user-defined functions. Thus, functions can be compiled one at a time and in any order. Moreover, previously compiled functions can be redefined (and then recompiled if desired), and the program can be rerun *without* having to recompile the functions that called them.

At any point in the development of a program, the compiler can be loaded and used as an **incremental compiler**. The function COMPILE is used to individually compile, or recompile, user-defined functions in the program.

Alternatively, the compiler can be used as a **file compiler**. The function COMPILE-FILE is used to automatically compile all the functions in a file, or list of files, and then delete the compiler. Although file compilation is not as flexible for program development as incremental compilation, it results in the most compact code and the most thorough elimination of the compiler from the **muLISP** data spaces.

The compiler is distributed as five files: COMPILE.LSP, COMPILE0.LSP, COMPILE1.LSP, COMPILE2.LSP, and COMPILE3.LSP. To load the compiler into **muLISP**, issue a command of the form

(LOAD *pathname*\\COMPILE)

where *pathname* specifies the drive and directory of the five compiler files. After a few seconds, the system will ask if you want to use the compiler as an incremental compiler or a file compiler.

If you choose to use it as an incremental compiler, the entire compiler is loaded and the functions COMPILE and DELETE-COMPILER (see Section 8.1) are available for use. Although the compiler can be loaded at any point in the development of a program, we recommend loading the compiler *after* the functions to be compiled are defined. This maximizes the amount of memory regained when the compiler is deleted.

If you choose to use it as a file compiler, only the function COMPILE-FILE is loaded (see Section 8.1). When COMPILE-FILE is subsequently called, it automatically loads the

remainder of the compiler *after* the file (or list of files) being compiled has been loaded. This maximizes the amount of memory regained when the compiler is deleted.

8.1 User Interface

This section documents the functions and macros that constitute the user interface to the **muLISP** compiler.

COMPILE [*name*] Function

If *name* is a user-defined eval function, (COMPILE *name*) translates the definition of *name* into assembly language instructions, assembles the instructions into 8086 machine code, loads the machine code into the **muLISP** code segment, and redefines *name* to invoke the machine code when called.

If *name* is not a user-defined eval function, COMPILE generates a Non-eval Function error break. No-eval functions, macros, and machine language routines can *not* be compiled (see Section 5.7.1).

However, the definition of *name* may contain all legitimate forms including:

- symbols
- numbers
- calls to machine language functions and special forms
- calls to user-defined eval and no-eval functions, spread or no-spread functions, compiled or uncompiled functions
- macro calls
- LAMBDA bodies applications such as those produced by the LET macro
- implicit PROGN and implicit COND constructs

All user-defined functions called directly in the definition of *name* must be defined, *though not necessarily compiled*, so the compiler can determine their type (see Section 5.3.3).

If COMPILE encounters no errors, it returns a list consisting of *name*, the number of bytes of machine code needed to compile *name*, and the entry point to the machine code as an offset in the **muLISP** code segment. If the compiler detects an error, a diagnostic error message is displayed and an error break occurs. If an error break occurs, *name* is *not* redefined.

The following example shows how the user-defined function RCONS is defined, compiled and then run. The code length and entry point returned by COMPILE will vary.

```
$ (DEFUN RCONS (X Y)                                    ;Define RCONS
    (CONS Y X) )
RCONS

$ (RCONS 1 2)                                           ;Test RCONS
(2 . 1)

$ (GETD 'RCONS)                                         ;LAMBDA definition of RCONS
(LAMBDA (X Y) (CONS Y X))

$ (COMPILE 'RCONS)                                      ;Compile RCONS
(RCONS 62 37457)

$ (GETD 'RCONS)                                         ;Address of machine code for RCONS
37457

$ (RCONS 1 2)                                           ;Retest RCONS
(2 . 1)
```

DELETE-COMPILER [ ] Macro

(DELETE-COMPILER) deletes the compiler from the **muLISP** environment by removing the definitions of all compiler functions, as well as the values and properties of certain compiler symbols. It returns the symbol DELETE-COMPILER, which is the result of removing its own definition using the function REMD.

The compiler takes up a significant amount of atom, cons, string, and code space (see Section 4.2.1). DELETE-COMPILER can be used to free this space when you have finished compiling functions. Deleting the compiler enables the garbage collector to reclaim most of the data space used by the compiler.

Since it is called automatically, the user need not call DELETE-COMPILER when compiling a file using COMPILE-FILE.

```
$ (RECLAIM)                                             ;Free bytes with compiler
311600

$ (DELETE-COMPILER)                                     ;Delete compiler
DELETE-COMPILER

$ (RECLAIM)                                             ;Free bytes w/o compiler
337767

$ (- * ***)                                             ;Bytes reclaimed
26167
```

COMPILE-FILE [*compile-files*, *load-files*] Function

If *compile-files* and *load-files* are file names *or* lists of file names, (COMPILE-FILE *compile-files load-files*) loads *compile-files* and *load-files*, loads the **muLISP** compiler, compiles the functions defined in *compile-files*, and deletes the compiler. As each function is compiled, COMPILE-FILE displays the function's name, its size in bytes, and its starting address in the **muLISP** code segment. COMPILE-FILE returns the number of bytes of free space remaining after the compilation.

Note that the files in *load-files* are loaded but *not* compiled. These files should contain function definitions for any external functions called by the functions to be compiled so that the compiler can determine the type of the functions being called.

If an error is detected while compiling a file, an error break will occur (see Section 8.4). Selecting the Continue option will continue the file compile beginning with the *next* function. Selecting the Abort option will abort compilation of the file and COMPILE-FILE will immediately return NIL.

```
$ (COMPILE-FILE 'GRAPHICS)                          ;Compile GRAPHICS.LSP
Loading: B:GRAPHICS.LSP
Loading: B:COMPILE0.LSP
Loading: B:COMPILE1.LSP
Loading: B:COMPILE2.LSP
Loading: B:COMPILE3.LSP
Compiling: B:GRAPHICS.LSP
SIN-DEG             (244 37457)                      ;Compilation data
COS-DEG             (146 37713)
   .                    .
   .                    .
   .                    .
Deleting compiler
320117

$ (SAVE 'GR)                                        ;Save compiled functions
```

8.2 Compilation Procedure

This section illustrates the basic procedure for using the compiler. It also shows the effect of compilation on the performance of some simple benchmark functions. You can do the examples on your computer exactly as they are shown here.

First bring up **muLISP**, and load COMMON.LSP to define BENCHMARK (see Section 5.8.7). However, do *not* load COMPILER.LSP at this time. Now define the functions APP and REV as follows:

```
$ (DEFUN APP (LST1 LST2)                                          ;Define APP
    ((NULL LST1)  LST2)
    (APP (REV (CDR (REV LST1))) (CONS (CAR (REV LST1)) LST2)) )
APP

$ (DEFUN REV (LST)                                                ;Define REV
    ((NULL LST)  NIL)
    (APP (REV (CDR LST)) (LIST (CAR LST))) )
REV

$ (REV NIL)                                                       ;Test REV and APP
NIL

$ (REV '(1))
(1)

$ (REV '(1 2 3))
(3 2 1)
```

APP and REV perform the same computation as the primitively defined functions APPEND and REVERSE, respectively. However, APP and REV perform their computations using a deliberately inefficient algorithm. As such, they provide good benchmarks of some basic elements of symbolic computation in LISP: list structure manipulation (CAR, CDR, CONS, and LIST), recursive function calls, and conditional branching.

Before compiling REV and APP, establish a timing baseline by running some simple timing tests on these functions:

```
$ (BENCHMARK (REV '(1 2 3 4 5 6 7)))                              ;Uncompiled
2.69 seconds
(7 6 5 4 3 2 1)

(BENCHMARK (REV '(1 2 3 4 5 6 7 8)))
10.76 seconds
(8 7 6 5 4 3 2 1)
```

Note that these timing tests were run on a PC AT compatible computer with 640K bytes of memory. Different hardware will produce different timing results.

Now load COMPILER.LSP as an incremental compiler as described in the introduction to this chapter. To see the effects of compiling, first compile and test REV by itself as follows:

```
$ (COMPILE 'REV)                                                    ;Compile REV
(REV 106 40241)

$ (BENCHMARK (REV '(1 2 3 4 5 6 7)))                                ;REV compiled
1.59 seconds
(7 6 5 4 3 2 1)

$ (BENCHMARK (REV '(1 2 3 4 5 6 7 8)))
6.21 seconds
(8 7 6 5 4 3 2 1)
```

Thus, compiling *just* REV has improved the overall execution speed by a factor of approximately 1.7. Of course, REV is still calling the *uncompiled* APP. Now compile APP and rerun the timing tests as follows:

```
$ (COMPILE 'APP)                                                    ;Compile APP
(APP 164 40369)

$ (BENCHMARK (REV '(1 2 3 4 5 6 7)))                                ;Both compiled
0.88 seconds
(7 6 5 4 3 2 1)

$ (BENCHMARK (REV '(1 2 3 4 5 6 7 8)))
3.68 seconds
(8 7 6 5 4 3 2 1)
```

Thus, compiling *both* REV and APP has improved the overall execution speed by a factor of approximately 3.0.

As indicated earlier, compiled functions can be redefined and subsequently recompiled *without* having to change or recompile other functions that called them. For example, suppose you replace (NULL LST1) with (ATOM LST1) in the definition of APP as follows:

```
$ (DEFUN APP (LST1 LST2)                                            ;Redefine APP
    ((ATOM LST1)  LST2)
    (APP (REV (CDR (REV LST1))) (CONS (CAR (REV LST1)) LST2)) )
APP

$ (BENCHMARK (REV '(1 2 3 4 5 6 7)))                                ;APP uncompiled
1.54 seconds
(7 6 5 4 3 2 1)

$ (BENCHMARK (REV '(1 2 3 4 5 6 7 8)))
6.21 seconds
(8 7 6 5 4 3 2 1)
```

As the increase in run time suggests, the previously compiled REV is now calling the new (and as yet *uncompiled*) definition of APP. Note that REV did *not* have to be recompiled.

You can now compile the new definition of APP as follows:

```
$ (COMPILE 'APP)                                          ;Recompile APP
(APP 162 40545)

$ (BENCHMARK (REV '(1 2 3 4 5 6 7)))                      ;Both compiled
0.88 seconds
(7 6 5 4 3 2 1)

$ (BENCHMARK (REV '(1 2 3 4 5 6 7 8)))
3.63 seconds
(8 7 6 5 4 3 2 1)
```

8.3 Performance Considerations

This section provides an overview of how the compiler improves the execution speed of **muLISP** programs. Sections 8.3.1 and 8.3.2 discuss the basic sources of improved execution speed: reducing the interpreter overhead and using in-line machine code for selected **muLISP** constructions. The net effects of these improvements are then discussed in Section 8.3.3.

8.3.1 Interpreter Overhead

While evaluating a function definition, the pseudo-code interpreter must work through the forms in the definition, analyzing each to determine how it should be evaluated (see Sections 5.6) and then evaluating it (see Section 5.7).

This analysis must be done *each* time an uncompiled function is called. The time required for the analysis is the overhead paid for the flexibility and convenience of interpretation. This analysis time is eliminated if the function is compiled.

While compiling a function definition, the compiler performs an analysis of the definition similar to that done by the interpreter. Based on this analysis, the compiler generates machine code to replace the function definition. In effect, this "hard-wires" the analysis of the function into machine code, and time need *not* be spent redoing the analysis each time the function is called.

However, since the pseudo-code interpreter is highly efficient, the gain in execution speed achieved by compiling functions is relatively modest (see Section 8.3.3).

8.3.2 In-line Machine Code

When a function definition is compiled, each of its forms are translated into machine code. Normally function calls of the form

$$(function\ form_1 \ldots form_n)$$

are translated into machine code that evaluates $form_1$ through $form_n$, pushes the results on the variable stack, and passes control to *function*.

However, calls to some primitives are compiled into in-line machine code. In-line code is much faster than calling the primitives as described above. Thus, primitives that are compiled in-line should be used to the maximum extent possible when writing functions that are to be compiled. The following primitives are compiled in-line:

Recognizers: SYMBOLP, ATOM, CONSP, NULL (see Section 5.1.1)

Comparators: EQ, NEQ (see Section 5.1.2)

Logical operators: NOT, AND, OR (see Section 5.1.3)

Selectors: CAR, CDR, CAAR, ..., CDDDDR (see Section 5.2.1)

Assignment: SETQ, PSETQ, POP, PUSH (see Section 5.3.1)

Control constructs: QUOTE, PROGN, LOOP, PROG1, IDENTITY, IF, COND, RETURN (see Section 5.6)

8.3.3 Speed Improvements

By reducing interpreter overhead and using in-line code, the compiler can significantly speed up programs. However, the compiler *cannot* reduce

- the time required to evaluate machine language functions that are not compiled in-line (see Section 8.3.2), in particular the arithmetic operators (see Section 5.4.3),
- the time required to perform a garbage collection (see Section 4.2.2), and
- the time required to read and write data from and to files and the console.

Thus, the net execution speed improvement of compiling a function will depend on the mix of particular functions and operations the function uses. For example, consider the following definitions for recursive and iterative versions of the factorial function:

```
(DEFUN RECURSIVE-FACTORIAL (N)
  ((ZEROP N)  1)
  (* N (RECURSIVE-FACTORIAL (SUB1 N))) )
```

```
(DEFUN ITERATIVE-FACTORIAL (N
    I)
  (SETQ I 1)
  (LOOP
    ((EQ N 0)  I)
    (SETQ I (* N I)
          N (SUB1 N)) ) )
```

This table shows the relative execution speed performance for various calls to these two functions, both uncompiled and compiled.

Function call	Uncompiled time (ms)	Compiled time (ms)	Speed-up factor
(RECURSIVE-FACTORIAL 0)	0.59	0.31	1.9
(RECURSIVE-FACTORIAL 1)	1.69	0.83	2.0
(RECURSIVE-FACTORIAL 10)	14.0	7.7	1.8
(RECURSIVE-FACTORIAL 100)	215.0	161.0	1.3
(ITERATIVE-FACTORIAL 0)	0.99	0.33	3.0
(ITERATIVE-FACTORIAL 1)	2.21	0.79	2.8
(ITERATIVE-FACTORIAL 10)	15.8	7.7	2.1
(ITERATIVE-FACTORIAL 100)	245.0	170.0	1.4

Note how the compiler speed-up factor *decreases* as RECURSIVE-FACTORIAL and ITERATIVE-FACTORIAL functions are applied to larger numbers. This is due to the longer time required for * to multiply larger numbers.

Also note how the *iterative* factorial function is improved more by compilation than the *recursive* one. This is because the iterative version uses more primitives that are compiled in-line (see Section 8.3.2).

8.4 Compiler Errors

Errors that occur during the compilation of a function are signaled by an error message that indicates the type of error and the function or form being compiled that caused the error. All errors cause the compilation to stop *without* changing the definition of the function being compiled. The two types of errors that can occur are compilation errors and internal errors.

8.4.1 Error Messages

Compilation errors can occur during the translation of a function definition into assembly language. Most compilation errors are caused by problems that would cause error breaks when encountered by the **muLISP** interpreter. Thus, a good way to avoid compilation errors is to test and debug functions *before* compiling them. The following paragraphs describe the compilation errors that may occur, showing the form of the error message as it is displayed on the console screen, and describing likely causes.

Non-eval Function Break: *name* indicates that *name* is not a user-defined eval function and hence can *not* be compiled. If *name* is a primitively defined function, a special form, an already compiled function, or a user-defined no-eval function or a macro, it cannot be compiled.

Nonsymbolic Argument Break: *form* indicates that *form* contains a nonsymbolic expression where a symbol is required. Typical forms are (SETQ ...), (PUSH ...), (POP ...), (INCQ ...), and so forth.

Undefined Function Call Break: *form* indicates that the car of *form* is not defined. If the car of *form* is a symbol, that symbol must be defined before compiling the function containing *form*.

Insufficient Code Space Break: *n* indicates that there is not enough room in the **muLISP** code segment for the *n* bytes of machine code produced in compiling a function. The code segment is *not* garbage collected, so if you have redefined and recompiled functions several times, it may be filled with useless machine code. In that case, you can start a fresh **muLISP**, load your program, load the compiler, and start over. If, however, your program is just too big to fit into the space available for compiled code, you may still wish to compile selected functions that you identify as being speed critical.

8.4.2 Internal Errors

Internal errors can occur at any point in the compilation process. They indicate some unforeseen problem with the **muLISP** compiler itself, *not* with the function you are compiling. Internal errors are displayed on the console screen in the following form:

```
Internal Error #iiii:j Break:
expr1
 ...
exprk
```

where *iiii:j* is the internal error number, and $expr_1$ through $expr_k$ characterize this particular occurrence of the error.

To get such errors fixed, report them to Soft Warehouse, Inc. using the *muLISP Problem Report* form provided at the back of this manual. Please include the entire internal error message, and the definition of the function that caused it.

8.5 Potential Problems

This section describes some potential problem areas in using the compiler, and presents means to minimize their effects.

8.5.1 Compiled Function Behavior

Compatible compilers transform function definitions into machine code that runs faster, returns the same values, and produces the same side-effects as the interpreted definitions. The **muLISP** compiler is compatible -- compiling a function should *not* change its basic computational behavior.

However, as is always the case with software, there may be some undiscovered bugs in the compiler that could cause compiled functions to return the wrong value or produce unwanted side-effects. Programs should always be thoroughly tested *after* compilation to make sure that they still behave properly. Please report any bugs that you find using the *muLISP Problem Report* form provided at the back of this manual. Include the definition of the function, and describe the difference in behavior before and after compilation.

Perfect compilers transform function definitions into machine code that runs faster, but otherwise act identically to the interpreted definitions. The **muLISP** compiler is not quite perfect -- compiling a function not only speeds it up, but also changes its behavior in some ways. These changes in behavior are necessary for compiled functions to achieve maximum execution speed.

The compiler does not insert console interrupt checks into compiled functions. Thus compiled functions cannot be interrupted unless they themselves check the keyboard for an interrupt character. However this is not a problem since console interrupts are normally required only when a program is being debugged. If they are necessary, do not compile functions that must be interruptible.

At compile time, all the functions to be compiled, as well as the functions they call, should not be traced (see Section 7.2). Compiling a traced function will simply compile the traced definition, not the function itself. Moreover, if a function being compiled calls a machine language primitive that is traced, there will be a smaller improvement in execution speed. This is because the traced primitive will *not* be recognized as a machine language routine, and will therefore be called *indirectly* using FUNCALL or APPLY rather than directly using a machine language call.

A compiled function can be traced using the debugger (see Section 7.2). Also, *user-defined* functions that the compiled function calls can be traced. However, tracing primitive functions and special forms that the compiled function calls will *not* cause a trace. Moreover, a traced compiled function will *not* cause a trace on a recursive call to itself. This is because calls to itself and calls to machine language primitives are made *directly* using a machine language call rather than indirectly using FUNCALL or APPLY.

Note that these complexities arising from the interaction of the debugger and the compiler are easily avoided by *not* compiling functions until after they have been thoroughly tested and debugged.

8.5.2 Name Conflicts

Besides the functions COMPILE, DELETE-COMPILER and COMPILE-FILE, the compiler uses and generates a variety of other symbols. If your program redefines or reassigns any compiler symbols, the resulting name conflicts may cause unpredictable results, both for the compiler and for your program.

To help avoid name conflicts, compiler symbols have been given distinctive patterns:

- Except for the three functions named above, all compiler functions, global and special variables, constants, property indicators, and CATCH tags end with !C.
- All constant symbols generated by the compiler to refer to numbers and quoted lists will have the form K!C*n*, for some integer *n*.
- All assembly language labels generated during compilation will have the form !C*n*, for some integer *n*.

Thus, to minimize the chance of name conflicts, simply avoid using the string !C in the function names, property indicators, and other symbols you use.

The following symbols should not be used as the names of functions to be compiled, since the compiler uses them for the names of 8086 registers and jump conditionals:

	AX	BX	CX	DX
Registers:	SI	DI	BP	SP
	CS	DS	ES	SS
	E	B	G	NE
Conditionals:	NB	GE	A	C
	L	NA	NC	LE

8.5.3 Redefining Primitives

The compiler uses many **muLISP** primitives and redefining these primitives can cause a variety of problems:

- Redefining primitives used by the compiler may cause it to malfunction. Such malfunctions may *not* be detected by its internal error checking.
- To enable incremental modification, compiled functions call user-defined functions *indirectly* using FUNCALL or APPLY. Therefore, redefining FUNCALL or APPLY may cause incorrect linkage from compiled code to other user-defined functions.
- To achieve maximum execution speed, compiled functions call primitive functions *directly* rather than going through FUNCALL or APPLY. Therefore, if a function that calls a primitive is compiled *before* the primitive is redefined, the compiled function will continue to call the original definition of the primitive.
- Primitives that are compiled in-line will *not* have the intended effect since the compiler will generate the usual in-line code for these primitives no matter what their definition (see Section 8.3.2).

The best way to avoid such problems is not to redefine any **muLISP** primitives in programs that are to be compiled.

If you must redefine some primitives, do it *before* compiling any functions in the program. This minimizes the potential for inconsistencies, since the compiler can distinguish functions defined in machine language from functions defined in pseudo-code. Thus, if a machine language function is redefined in pseudo-code, calls to the function will be compiled correctly.

Chapter 9. Machine-coded Routines

Some users may find it necessary to extend **muLISP** by defining some routines in machine code instead of pseudo-code. For the most part, the communication primitives (see Section 5.8) and the compiler (see Chapter 8) eliminate the need to program in assembly language. However, user-defined machine-coded routines may be required for the following reasons:

- to control some special computer hardware or peripheral device;
- to communicate with the host operating system; or
- to implement speed-critical functions more efficiently.

This chapter describes how to write, load, and link functions and special forms defined in machine code. It assumes you are familiar with the representation of data objects (see Section 4.1), the memory management system (see Section 4.2), and the architecture and assembly language of the Intel 8086 microprocessor.

The **base page** contains the default values for several system constants (e.g. computer type, maximum program size, etc.), the default readtable, the macro character function table, the pseudo-registers, the line editor control key table, and the special form jump table. The base page begins at 100H in the code segment.

Appendix H is an assembly language listing that shows the offset addresses of the constants in the base page. Programs can get the value of these constants using CSMEMORY (see Section 5.8.7). Machine-coded routines can get their value using offsets in the code segment.

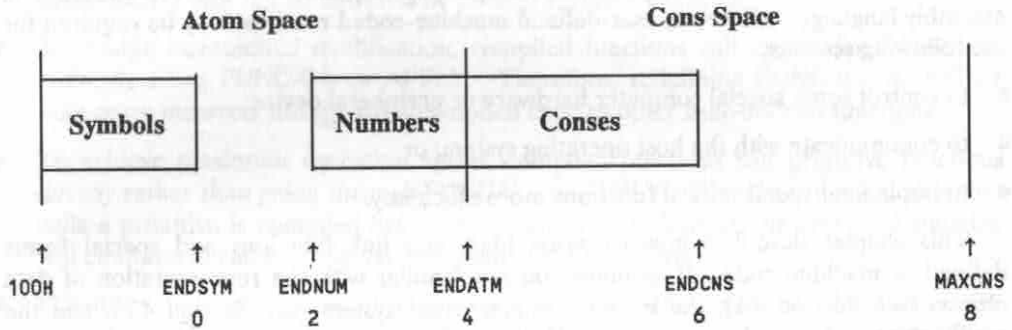
The base page is the first 300H (hexadecimal) bytes of MULISP.COM. You can use the program DEBUG.COM (see the MS-DOS manual) to permanently change the default value of constants in the base page. For example, this is how to set the computer type number at offset 0357H to 2 (the PC compatible type number):

```
DEBUG MULISP.COM
-E357
221A:0357 00.2
-W
-Q
```

The **global storage area** contains the variables that point to the ends of the various data spaces and other system variables. Appendix I is an assembly language listing that shows the offset address of these variables in the data segment. Programs can determine the value of these variables using DSMEMORY (see Section 5.8.7). Machine-coded routines can determine their value using offsets in the data segment.

9.1 Data Representation

In order to write machine language routines, you need to understand how and where data objects are stored in memory. The following diagram illustrates where the three types of objects are stored:



The diagram also gives the names of the variables that point to the current end of the various regions. These pointers are maintained in the global storage area at the offsets indicated under their name. As an efficiency measure, the register DX always contains the same value as ENDATM.

Using these variables, it is easy for a machine language routine to determine the type of a data object (a symbol, a number, or a cons) merely from its address. For example, if SI points to the object, the following code jumps to one of three routines based upon the object's type:

```
ENDSYM  EQU     0
ENDNUM  EQU     2
ENDATM  EQU     4

        CMP     SI,WORD PTR DS:[ENDSYM]
        JB      LABEL1                  ;Jump if SI points to a symbol
        CMP     SI,DX                   ;Note that DX = ENDATM
        JB      LABEL2                  ;Jump if SI points to a number
        JMP     LABEL3                  ;SI points to a cons
```

Unfortunately the 8086 divides its 20 bit address space into **segments** that can only be a maximum of 64K (kilobytes) each. The pointer cells making up a data object are stored at the same offset in different segments. Its car cell is stored in the segment pointed to by the DS segment register. Its cdr cell is stored in the segment pointed to by the ES segment register. If the object is an atom it has a third cell and a fourth cell that are stored in the definition and print name pointer segments respectively. The 16 bit paragraph addresses of these segments are stored in the variables DEFSEG and PNPSEG in the global storage area.

Knowing this, it is easy for machine language routines to access the contents of an object's cells. For example, if SI points to a symbol, the following code loads a pointer to its value into AX, a pointer to its property list into BX, a pointer to its function definition into CX, and a pointer to its print name into DI:

```
DEFSEG  EQU     10H
PNPSEG  EQU     12H

        MOV     AX,[SI]                 ;AX points to symbol's value
        MOV     BX,ES:[SI]              ;BX points to symbol's property list
        PUSH    DS
        MOV     DS,WORD PTR DS:[DEFSEG]
        MOV     CX,[SI]                 ;CX points to symbol's definition
        POP     DS
        PUSH    DS
        MOV     DS,WORD PTR DS:[PNPSEG]
        MOV     DI,[SI]                 ;DI points to symbol's print name
        POP     DS
```

Since the symbol's value and property list must be *bona fide* data objects, they can be accessed as described earlier. If the definition pointer in CX is odd, it contains the offset of a machine language routine in the code segment (CS). If CX is even, it contains the offset of pseudo-code that is stored in the stack segment (SS). The print name pointer in DI contains the offset of the ASCII print name string that is stored in the print name string segment. The paragraph address of the print name string segment is stored in the variable PNSSEG in the global storage area.

The above code can also be used to get the contents of the four cells comprising a number. The vector cell of a big integer contains the offset of the number vector that is stored in the number vector segment. The paragraph address of the number vector segment is stored in the variable VECSEG in the global storage area.

9.2 Argument Passing

Both functions or special forms (see Section 5.7.1) can be defined as machine-coded routines. The entry point address of functions *must* be odd. The entry point address of special forms can be even *or* odd since they are only referenced indirectly via the special form jump table in the base page.

As the arguments to a function are evaluated, the addresses of the resulting data objects are pushed on top of the variable stack. The BP register is the variable stack pointer. Then the machine-coded function is called with the arguments stored above the top of the variable stack. On entry to the routine, CX contains two times the number of actual arguments that were in the call to the function.

If the following routine is called at the beginning of a machine-coded function, the values of the function's first three arguments are stored in the SI, DI, and BX registers respectively. If there are not three arguments, the extra registers are set to NIL:

```
NIL     EQU     100H                    ;The address of NIL is a constant

GETTHR: CMP     CX,6
        JB      GETTWO                  ;Jump if less than 3 arguments
        MOV     SI,[BP]                 ;SI: argument one
        MOV     DI,[BP+2]               ;DI: argument two
        MOV     BX,[BP+4]               ;BX: argument three
        RET

GETTWO: MOV     BX,NIL                  ;BX: NIL
        CMP     CX,4
        JB      GETONE                  ;Jump if less than 2 arguments
        MOV     SI,[BP]                 ;SI: argument one
        MOV     DI,[BP+2]               ;DI: argument two
        RET

GETONE: MOV     DI,NIL                  ;DI: NIL
        JCXZ    GETZRO                  ;Jump if no arguments
        MOV     SI,[BP]                 ;SI: argument one
        RET

GETZRO: MOV     SI,NIL                  ;SI: NIL
        RET
```

Since the arguments to a special form are not evaluated, a list of the actual arguments is passed to the special form routine in the SI register. Special forms can be invoked either by evaluating pseudo-code or by evaluating a data object. Thus the actual argument list pointed to by SI can be either pseudo-code or a *bona fide* data object.

Separate routines must be written to handle both cases. The offset addresses of the two special form routines are then stored in a two entry jump table. Space for 10 user-defined special forms has been set aside in the base page. The address of the pseudo-

code handler is stored in the first entry; the address of the *bona fide* data object handler in the second.

Both machine-coded functions and special forms return their value in the DI register. DI must contain the offset address of a *bona fide* data object. If the routine does not need to return a value, DI should still be set to 100H, the offset of NIL.

9.3 Calling Machine-coded Routines

User-defined routines can call the large number of functions and special forms already defined in machine code. These routines use the same **calling convention** to receive arguments and return values as that described for user-defined routines. The offset address in the code segment of the functions can be determined using GETD (see Section 5.3.3):

```
$ (SETQ *PRINT-BASE* 16)                          ;Print in hexadecimal
10

$ (GETD 'CONS)                                    ;Get CONS's entry point
0B27                                                    ;Result may vary

$ (GETD 'LOOP)                                    ;Get LOOP's jump table
1B8                                                     ;Result may vary

$ (SETQ *PRINT-BASE* 10)                          ;Print in decimal
10
```

Before calling other machine-coded routines or when returning from your routine, ensure that DX and the four segment registers are restored to their original value.

Calls to machine language routines that do consing or generate numbers can invoke the garbage collector. Garbage collections will destroy unreferenced data objects and can change the address of referenced objects. To ensure that pointers to objects are saved and updated during a garbage collection, they must be pushed onto the variable stack. The BP register is the variable stack pointer. The following code segment illustrates how the pointer in BX is pushed on and later popped off the variable stack during a call to CONS:

```
CONS    EQU     0B27H

        MOV     [BP],BX                 ;Push BX on BP stack
        INC     BP                      ;Increment BP stack pointer
        INC     BP
        MOV     AX,OFFSET CONS+3
        EVEN                            ;Force CALL to an odd address
        NOP
        CALL    AX                      ;Set DI to CONS (SI, DI)
        DEC     BP                      ;Decrement BP stack pointer
        DEC     BP
        MOV     BX,[BP]                 ;Pop updated BX from BP stack
```

Note the use of the EVEN and NOP opcodes immediately preceding the call to CONS. The assembly language instruction

```
CALL AX
```

is a two byte instruction. Therefore, if the call instruction begins on an odd address, the return address that is pushed on the SP stack will be odd.

During a reallocating garbage collection, references on the SP stack to pseudo-code must be updated when the pseudo-code is moved during reallocation. To distinguish pseudo-code pointers from return addresses on the SP stack, the reallocator relies on the fact that pseudo-code pointers are even, whereas return addresses are odd. Therefore, it is necessary to ensure that a return address (or anything else for that matter) that is pushed on the SP stack is odd *if* a garbage collection can occur while the return address is on the stack.

9.4 Loading and Linking

Use the following three step process to load and link machine-coded routines:

- Call ALLOCATE to set aside enough space in the code segment for the routine.
- Call BINARY-LOAD to load the routine into memory.
- Call PUTD to link a symbol's function definition cell to the routine.

If *n* is a positive integer, (ALLOCATE *n*) frees *n* bytes of memory in the code segment and returns the offset of the base of the newly allocated memory. Note that if ALLOCATE returns an integer, it will be even.

If *offset* is an integer between 0 and 65535 inclusive and the file on the drive and directory specified by *filename* exists, (BINARY-LOAD *filename offset*) loads *filename* into the **muLISP** code segment at *offset* and returns the number of bytes loaded. If no file name extension is specified in *filename*, BINARY-LOAD assumes a file name extension of BIN. Normally the *offset* in a call to BINARY-LOAD is the offset address returned by the call to ALLOCATE plus 1 to make it odd.

If *name* is a symbol and *n* is the odd entry address (i.e. its offset in the code segment) of a user-defined machine language function routine, (PUTD *name n*) links *name* to the function. If *n* is the even offset address of a special form jump table in the base page, (PUTD *name n*) links *name* to the special form routines pointed to by the table.

The following function (LOAD-LINK *name filename bytes*) is a useful utility that automates the process for loading and linking user-defined machine language routines:

```
(DEFUN LOAD-LINK (NAME FILE-NAME BYTES
    OFFSET )
  ((SETQ OFFSET (ALLOCATE BYTES))                          ;Allocate memory
    ((BINARY-LOAD FILE-NAME (ADD1 OFFSET))                      ;Load file
      (PUTD NAME (ADD1 OFFSET)) ) ) )                        ;Link function
```

As an example of the major points discussed in this chapter, consider the problem of converting the following user-defined function into a machine language routine:

```
(DEFUN COPY (OBJ)
  ((ATOM OBJ)  OBJ)
  (CONS (COPY (CAR OBJ)) (COPY (CDR OBJ))) )
```

COPY is equivalent to the primitive function COPY-TREE. Appendix J is a listing of the equivalent 8086 assembly language routine for COPY. If the routine is assembled and saved as the file COPY.BIN on drive B, it can be loaded and linked to the symbol COPY as follows:

```
$ (LOAD-LINK COPY B:COPY 60)                               ;Load and link COPY
COPY

$ (COPY '(A (B . C) D))                                    ;Call the new function COPY
(A (B . C) D)
```

Chapter 10. LISP References

The **muLISP** tutorial system provides an on-line introduction to LISP programming (see Section 3.2). To learn more advanced techniques, you will need to consult the extensive body of literature on LISP and artificial intelligence (AI).

This chapter lists many of the relevant professional societies, periodicals, and books. They can serve as a starting point for your explorations into this exciting area of software research.

10.1 Professional Societies

The American Association for Artificial Intelligence (AAAI), 445 Burgess Drive, Menlo Park, CA, 94025; phone (415) 328-3123.

The ACM Special Interest Group on Artificial Intelligence (SIGART), Association for Computing Machinery, 1133 Avenue of the Americas, New York, NY, 10036.

10.2 Periodicals

AI & Society, Springer-Verlag New York Inc., Service Center Secaucus, 44 Hartz Way, Secaucus, NJ, 07094; phone (201) 348-4033.

AI Expert, Miller Freeman Publications, 500 Howard St., San Francisco, CA, 94105; phone (415) 397-1881.

AI Magazine, American Association for Artificial Intelligence, 445 Burgess Drive, Menlo Park, CA, 94025; phone (415) 328-3123.

AI Today, 104 Frame Road, Elkview, WV, 25071; phone (304) 965-5548.

AIWeek, Applied Artificial Intelligence Reporter, University of Miami Intelligent Computer Systems Research Institute, P.O. Box 248235, 421C Jenkins Building, Coral Gables, FL, 33124; phone (305) 284-5161.

Artificial Intelligence, P.O. Box 211, 1000 AE Amsterdam, The Netherlands.

Expert Systems, Learned Information (Europe) Ltd., Woodside, Hinksey Hill, Oxford OX1 5AU, England; phone Oxford (0865) 730275; telex 837704 INFORM G. *or* Learned Information Inc., 143 Old Marlton Pike, Medford, NJ, 08055, USA; phone (609) 654-6266.

Expert Systems User, Expert Systems User Ltd., 52-54 Southwark Street, London SE1 1UN, Great Britain; phone 01-407 7400; telex 9413548.

IEEE Expert, IEEE Computer Society, 345 East 47th St., New York, NY, 10017.

LISP and Symbolic Computation, Kluwer Academic Publishers, P.O. Box 358, Accord Station, Hingham, MA, 02018-9990.

LISP Pointers, Mary S. Van Deusen, Editor, IBM Watson Research, P.O. Box 704, Yorktown Heights, NY, 10598.

Machine Intelligence, Ellis Horwood Ltd., Market Cross House, Cooper St., Chichester, West Sussex, PO19 1EB, England.

PC AI, Knowledge Technology, Inc., 3310 West Bell Rd., Ste. 119, Phoenix, AZ, 85023; phone (602) 439-3253.

Pattern Recognition, Pergamon Press Inc., Maxwell House, Fairview Park, Elmsford, NY, 10523.

10.3 Books

Allen, J. R., **Anatomy of LISP**, McGraw-Hill Book Company, New York, NY, 1978.

Anderson, J. R., Corbett, A. T., and Reiser, B. J., **Essential LISP**, Addison-Wesley Publishing Company, Inc., Reading, MA, 1987.

Barr, A., and Feigenbaum, E. A., (eds), **The Handbook of Artificial Intelligence**, William Kaufmann, Inc., Los Altos, CA, 1982.

Berk, A. A., **LISP, The Language of Artificial Intelligence**, Van Nostrand Reinhold Company, New York, NY, 1985.

Berkeley, E. C., and Bobrow, D. G., (eds), **The Programming Language LISP: Its Operation and Applications**, The M.I.T. Press, Cambridge, MA, 1964.

Bromley, H., **LISP Lore: A Guide to Programming the LISP Machine**, Kluwer Academic Publishers, Hingham, MA, 1986.

Brookes, R. A., **Programming in Common LISP**, John Wiley & Sons, New York, NY, 1985.

Charniak, E., et al, **Artificial Intelligence Programming**, Lawrence Erlbaum Associates, 365 Broadway, Hillsdale, NJ, 1980.

Coxhead, P., **Starting LISP for AI**, Blackwell Scientific Publications, Oxford, England, 1987.

Doukidis, G. I., et al, **LISP: From Foundations to Applications**, Chartwell-Bratt Ltd., Kent, England, 1988.

Franz Inc., **Common LISP, The Reference**, Addison-Wesley Publishing Company, Inc., Reading, MA, 1988.

Friedman, D. P., et al, **The Little LISPer, Second Edition**, Science Research Associates, Inc., Chicago, IL, 1986.

Gabriel, R. P., **Performance and Evaluation of LISP Systems**, The M.I.T. Press, Cambridge, MA, 1985.

Hasemer, T., **Looking at LISP**, Addison-Wesley Publishing Co., Reading, MA, 1984.

Henderson, P., **Functional Programming: Application and Implementation**, Prentice-Hall, Englewood Cliffs, NJ, 1980.

Keene, S. E., **Object-Oriented Programming in Common LISP**, Addison-Wesley Publishing Company, Reading, MA, 1989.

Kessler, R. R., **LISP, Objects, and Symbolic Programming**, Scott, Foresman and Company, Glenview, IL, 1988.

McCarthy, J., **Recursive Functions of Symbolic Expressions and Their Computation by Machine**, Communications of the ACM, pages 184-195, April 1960.

McCarthy, J., et al, **LISP 1.5 Programmer's Manual**, M.I.T. Press, Cambridge, MA, 1963.

Maurer, W. D., **A Programmer's Introduction to LISP**, American Elsevier, New York, NY, 1973.

Mueller, R. A., et al, **Symbolic Computing with LISP and Prolog**, John Wiley & Sons, New York, NY, 1988.

Queinnec, C., **LISP**, John Wiley & Sons, New York, NY, 1983.

Schoffa, G., **Die Programmiersprache LISP**, Franzis-Verlag GmbH, Munchen, West Germany, 1987.

Siklossy, L., **Let's Talk LISP**, Prentice-Hall, Englewood Cliffs, NJ, 1976.

Steele, G. L. Jr., **Common LISP: The Language**, Digital Press, Burlington, MA, 1984.

Touretzky, D. S., **LISP: A Gentle Introduction to Symbolic Computation**, Harper and Row, New York, NY, 1983.

Weinreb, D. L. & Moon, D., **LISP Machine Manual**, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1981.

Weissman, C., **LISP 1.5 Primer**, Dickenson Publishing Co., Belmont, CA, 1968.

Wilensky, R., **Common LISPcraft**, W. W. Norton & Company, New York, NY, 1986.

Winston, P. H. & Horn, B. K. P., **LISP, Second Edition**, Addison-Wesley Publishing Co., Reading, MA, 1984.

Appendix A. PC Video Modes

This table summarizes the number of colors and resolutions provided by each of the video modes on PC compatible computers (see Section 5.8.6).

Video Mode	Mode Type	Fore Colors	Back Colors	Char Resolution	Graphics Resolution	Adapter Card	Monitor/Computer Type
0	text	16	8	40 x 25	40 x 25	CGA	Color
1	text	16	8	40 x 25	40 x 25	CGA	Color
2	text	16	8	80 x 25	80 x 25	CGA	Color
3	text	16	8	80 x 25	80 x 25	CGA	Color
4	graphics	4	4	40 x 25	320 x 200	CGA	Color
5	graphics	4	4	40 x 25	320 x 200	CGA	Color
6	graphics	2	2	80 x 25	640 x 200	CGA	Color
7	text	3	2	80 x 25	80 x 25	MDA	Monochrome
8	graphics	16	16	20 x 25	160 x 200	PCjr	Color
9	graphics	16	16	40 x 25	320 x 200	PCjr	Color
10	graphics	4	4	80 x 25	640 x 200	PCjr	Color
13	graphics	16	16	40 x 25	320 x 200	EGA	Color
14	graphics	16	16	80 x 25	640 x 200	EGA	Color
15	graphics	3	2	80 x 25	640 x 350	EGA	Monochrome
16	graphics	16	16	80 x 25	640 x 350	EGA	Enhanced
17	graphics	2	2	80 x 30	640 x 480	MCGA	Multi-sync
18	graphics	16	16	80 x 30	640 x 480	VGA	Multi-sync
19	graphics	256	256	40 x 25	320 x 200	MCGA	Multi-sync
32	graphics	2	2	40 x 16	240 x 128	HP	95LX
64	graphics	2	2	80 x 25	640 x 400	AT&T	6300
65	graphics	16	16	80 x 25	640 x 200	AT&T	6300
66	graphics	16	16	80 x 25	640 x 400	AT&T	6300
116	graphics	16	16	80 x 25	640 x 400	Toshiba	T3100
126	graphics	2	2	90 x 43	720 x 348	Hercules	Monochrome
127	graphics	2	2	90 x 29	720 x 348	Hercules	Monochrome

Adapter cards: Color Graphics Adapter (CGA), Monochrome Display Adapter (MDA), Enhanced Graphics Adapter (EGA), Multi-Color Graphics Array adapter (MCGA), Video Graphics Array adpater (VGA), Hercules compatible graphics card (Hercules).

Monitor types: RGB Color monitor (Color), Monochrome monitor (Monochrome), Enhanced graphics monitor (Enhanced), Multi-sync graphics monitor (Multi-sync), HP 95LX Palmtop computer, AT&T 6300 computer, Toshiba T3100 computer.

Appendix B. PC Screen Colors

These tables summarize the colors and their associated color numbers that are available in each of the video modes on PC compatible computers.

Video modes 0, 1, 2, 3, 8, 9, 13, 14, 16, and 18

Color Number	Color	Color Number	Color
0	black	8	dark gray
1	blue	9	light blue
2	green	10	light green
3	cyan	11	light cyan
4	red	12	light red
5	magenta	13	light magenta
6	brown	14	yellow
7	light gray	15	white

In all these video modes, the foreground color can be set to any one of these 16 colors. In the graphics video modes (8, 9, 13, 14, 16, and 18), the background color can also be set to any one to these 16 colors. However, in the text video modes (0, 1, 2, and 3), setting the background color to a number greater than 7 results in a background color of between 0 and 7 and the blinking of characters.

Video modes 4, 5, and 10

Color Number	Palette 0 Color	Palette 1 Color
0	border	border
1	green	cyan
2	red	magenta
3	yellow	white

The color displayed for color number 0 can be instantly changed to any one of 16 different colors by changing the border color.

Video modes 6, 15, and 17

Color Number	Color
0	black
1	border

The color displayed for color number 1 can be instantly changed to any one of 16 different colors by changing the border color.

Video mode 7

Foreground	Background	Intensity	Underline	Blinking
0	0	Blank	No	No
1	0	Low	Yes	No
7	0	Low	No	No
9	0	High	Yes	No
15	0	High	No	No
0	7	Inverse	No	No
0	15	Inverse	No	Yes
1	15	Low	Yes	Yes
7	15	Low	No	Yes
9	15	High	Yes	Yes
15	15	High	No	Yes

Video mode 7 is for monochrome display adapters and monitors. Instead of different colors, this mode displays characters with different character attributes when the foreground and background are changed. Characters can be displayed in low or high intensity, or normal or inverse video, underlined or not, and blinking or not.

Appendix C. PC Screen Characters

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00H	NUL	32	20H	SP	64	40H	@	96	60H	`
1	01H	☺	33	21H	!	65	41H	A	97	61H	a
2	02H	☻	34	22H	"	66	42H	B	98	62H	b
3	03H	♥	35	23H	#	67	43H	C	99	63H	c
4	04H	♦	36	24H	$	68	44H	D	100	64H	d
5	05H	♣	37	25H	%	69	45H	E	101	65H	e
6	06H	♠	38	26H	&	70	46H	F	102	66H	f
7	07H	• BEL	39	27H	'	71	47H	G	103	67H	g
8	08H	◘ BS	40	28H	(	72	48H	H	104	68H	h
9	09H	○ TAB	41	29H	)	73	49H	I	105	69H	i
10	0AH	■ LF	42	2AH	*	74	4AH	J	106	6AH	j
11	0BH	♂	43	2BH	+	75	4BH	K	107	6BH	k
12	0CH	♀	44	2CH	,	76	4CH	L	108	6CH	l
13	0DH	♪ RET	45	2DH	-	77	4DH	M	109	6DH	m
14	0EH	♫	46	2EH	.	78	4EH	N	110	6EH	n
15	0FH	☼	47	2FH	/	79	4FH	O	111	6FH	o
16	10H	►	48	30H	0	80	50H	P	112	70H	p
17	11H	◄	49	31H	1	81	51H	Q	113	71H	q
18	12H	↕	50	32H	2	82	52H	R	114	72H	r
19	13H	‼	51	33H	3	83	53H	S	115	73H	s
20	14H	¶	52	34H	4	84	54H	T	116	74H	t
21	15H	§	53	35H	5	85	55H	U	117	75H	u
22	16H	▬	54	36H	6	86	56H	V	118	76H	v
23	17H	↨	55	37H	7	87	57H	W	119	77H	w
24	18H	↑	56	38H	8	88	58H	X	120	78H	x
25	19H	↓	57	39H	9	89	59H	Y	121	79H	y
26	1AH	→	58	3AH	:	90	5AH	Z	122	7AH	z
27	1BH	←	59	3BH	;	91	5BH	[	123	7BH	{
28	1CH	∟	60	3CH	<	92	5CH	\	124	7CH	|
29	1DH	↔	61	3DH	=	93	5DH	]	125	7DH	}
30	1EH	▲	62	3EH	>	94	5EH	^	126	7EH	~
31	1FH	▼	63	3FH	?	95	5FH	_	127	7FH	⌂

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80H	Ç	160	A0H	á	192	C0H	└	224	E0H	α
129	81H	ü	161	A1H	í	193	C1H	┴	225	E1H	ß
130	82H	é	162	A2H	ó	194	C2H	┬	226	E2H	Γ
131	83H	â	163	A3H	ú	195	C3H	├	227	E3H	π
132	84H	ä	164	A4H	ñ	196	C4H	─	228	E4H	Σ
133	85H	à	165	A5H	Ñ	197	C5H	┼	229	E5H	σ
134	86H	å	166	A6H	ª	198	C6H	╞	230	E6H	µ
135	87H	ç	167	A7H	º	199	C7H	╟	231	E7H	τ
136	88H	ê	168	A8H	¿	200	C8H	╚	232	E8H	Φ
137	89H	ë	169	A9H	⌐	201	C9H	╔	233	E9H	Θ
138	8AH	è	170	AAH	¬	202	CAH	╩	234	EAH	Ω
139	8BH	ï	171	ABH	½	203	CBH	╦	235	EBH	δ
140	8CH	î	172	ACH	¼	204	CCH	╠	236	ECH	∞
141	8DH	ì	173	ADH	¡	205	CDH	═	237	EDH	φ
142	8EH	Ä	174	AEH	«	206	CEH	╬	238	EEH	ε
143	8FH	Å	175	AFH	»	207	CFH	╧	239	EFH	∩
144	90H	É	176	B0H	░	208	D0H	╨	240	F0H	≡
145	91H	æ	177	B1H	▒	209	D1H	╤	241	F1H	±
146	92H	Æ	178	B2H	▓	210	D2H	╥	242	F2H	≥
147	93H	ô	179	B3H	│	211	D3H	╙	243	F3H	≤
148	94H	ö	180	B4H	┤	212	D4H	╘	244	F4H	⌠
149	95H	ò	181	B5H	╡	213	D5H	╒	245	F5H	⌡
150	96H	û	182	B6H	╢	214	D6H	╓	246	F6H	÷
151	97H	ù	183	B7H	╖	215	D7H	╫	247	F7H	≈
152	98H	ÿ	184	B8H	╕	216	D8H	╪	248	F8H	°
153	99H	Ö	185	B9H	╣	217	D9H	┘	249	F9H	∙
154	9AH	Ü	186	BAH	║	218	DAH	┌	250	FAH	·
155	9BH	¢	187	BBH	╗	219	DBH	█	251	FBH	√
156	9CH	£	188	BCH	╝	220	DCH	▄	252	FCH	ⁿ
157	9DH	¥	189	BDH	╜	221	DDH	▌	253	FDH	²
158	9EH	₧	190	BEH	╛	222	DEH	▐	254	FEH	■
159	9FH	ƒ	191	BFH	┐	223	DFH	▀	255	FFH	

Appendix D. PC Extended Function Keys

The **extended function keys** on PC compatible computers include the ten function keys located on the left or top of the keyboard, the arrow keys located on the right side of the keyboard, and the **Alt** key used in conjunction with other keys.

Programs can detect the pressing of an extended function key if READ-BYTE returns a 255. This table summarizes the *next* byte that READ-BYTE will return for each extended function key:

Byte	Key	Byte	Key	Byte	Key	Byte	Key
15	Shift-Tab	48	Alt-B	85	Shift-F2	109	Alt-F6
16	Alt-Q	49	Alt-N	86	Shift-F3	110	Alt-F7
17	Alt-W	50	Alt-M	87	Shift-F4	111	Alt-F8
18	Alt-E	59	F1	88	Shift-F5	112	Alt-F9
19	Alt-R	60	F2	89	Shift-F6	113	Alt-F10
20	Alt-T	61	F3	90	Shift-F7	114	Ctrl-PrtSc
21	Alt-Y	62	F4	91	Shift-F8	115	Ctrl ←
22	Alt-U	63	F5	92	Shift-F9	116	Ctrl →
23	Alt-I	64	F6	93	Shift-F10	117	Ctrl-End
24	Alt-O	65	F7	94	Ctrl-F1	118	Ctrl-PgDn
25	Alt-P	66	F8	95	Ctrl-F2	119	Ctrl-Home
30	Alt-A	67	F9	96	Ctrl-F3	120	Alt-1
31	Alt-S	68	F10	97	Ctrl-F4	121	Alt-2
32	Alt-D	71	Home	98	Ctrl-F5	122	Alt-3
33	Alt-F	72	↑	99	Ctrl-F6	123	Alt-4
34	Alt-G	73	PgUp	100	Ctrl-F7	124	Alt-5
35	Alt-H	75	←	101	Ctrl-F8	125	Alt-6
36	Alt-J	77	→	102	Ctrl-F9	126	Alt-7
37	Alt-K	79	End	103	Ctrl-F10	127	Alt-8
38	Alt-L	80	↓	104	Alt-F1	128	Alt-9
44	Alt-Z	81	PgDn	105	Alt-F2	129	Alt-0
45	Alt-X	82	Ins	106	Alt-F3	130	Alt -
46	Alt-C	83	Del	107	Alt-F4	131	Alt-=
47	Alt-V	84	Shift-F1	108	Alt-F5	132	Ctrl-PgUp

Appendix E. Line Editor Commands

This table summarizes the editing commands available when using the **muLISP** line editor.

Control key	PC key	Cursor control commands
Ctrl-D	→	move right a character
Ctrl-S	←	move left a character
Ctrl-F	Ctrl →	move right a word
Ctrl-A	Ctrl ←	move left a word
Ctrl-Q D	End	move to right end of line
Ctrl-Q S	Home	move to left end of line
Ctrl-I	Tab	move right to next tab stop
		Deleting text commands
Ctrl-G	Del	delete character under cursor
Ctrl-H	Backspace	delete character left of cursor
Rub	Ctrl-Backspace	delete character left of cursor
Ctrl-T		delete word under cursor
Ctrl-Y		delete line of text
Ctrl-Q Y		delete right end of line
Ctrl-Q H	Ctrl-Q Backspace	delete left end of line
Ctrl-Q Rub		delete left end of line
Ctrl-U	F3	undelete last deleted line
		Entering text commands
Ctrl-V	Ins	text entry mode (insert or overwrite) toggle
Ctrl-P		enter an escape character
Ctrl-M	Enter	quit line editor, return line of text
Ctrl-J	Ctrl-Enter	quit line editor, return line of text
Ctrl-[	Esc	quit line editor, quit read-eval-print loop

Appendix F. WordStar-like Editor Commands

This table summarizes the edit mode commands available when the **muLISP** editor is WordStar-like. The commands are divided into the same categories used in Section 7.1.

Control key	PC key	Cursor control commands
Ctrl-D	→	move forward a character
Ctrl-S	←	move backward a character
Ctrl-F	Ctrl →	move forward a word
Ctrl-A	Ctrl ←	move backward a word
Ctrl-X	↓	move down a line
Ctrl-E	↑	move up a line
Ctrl-Q D	End	move to right end of line
Ctrl-Q S	Home	move to left end of line
Ctrl-Q X	Ctrl-End	move to bottom row of screen
Ctrl-Q E	Ctrl-Home	move to top row of screen
Ctrl-I	Tab	move right to next tab stop (overwrite mode)
Ctrl-Q I	Shift-Tab	move left to previous tab stop
Ctrl-M	Enter	move to start of next line (overwrite mode)
Ctrl-J	Ctrl-Enter	move down line and indent
Ctrl-Q P		move to last position
Ctrl-K *n*		mark position *n* (*n* = 0, 1, 2, or 3)
Ctrl-Q *n*		move to position *n* (*n* = 0, 1, 2, or 3)
		Display control commands
Ctrl-Z		scroll text up a line
Ctrl-W		scroll text down a line
Ctrl-C	PgDn	scroll text up a screen
Ctrl-R	PgUp	scroll text down a screen
Ctrl-Q C	Ctrl-PgDn	move to end of text
Ctrl-Q R	Ctrl-PgUp	move to start of text
		Entering text commands
Ctrl-V	Ins	switch text entry mode (insert or overwrite)
Ctrl-I	Tab	space over to next tab stop (insert mode)
Ctrl-M	Enter	insert a new line after cursor (insert mode)
Ctrl-N		insert a new line before cursor
Ctrl-P		enter an escape character

Control key	PC key	Deleting text commands
Ctrl-G	Del	delete character under cursor
Ctrl-H	Backspace	delete character left of cursor
Rub	Ctrl-Backspace	delete character left of cursor
Ctrl-T		delete word under cursor
Ctrl-Q T		delete word left of cursor
Ctrl-Y		delete line under cursor
Ctrl-Q Y		delete right end of line
Ctrl-Q H	Ctrl-Q Backspace	delete left end of line
Ctrl-Q Rub		delete left end of line
Ctrl-U		restore last deleted text
		Block commands
Ctrl-O B	Ctrl-K B	mark beginning of block
Ctrl-O E	Ctrl-K K	mark end of block
Ctrl-O M	Ctrl-K V	move block to cursor
Ctrl-O C	Ctrl-K C	copy block at cursor
Ctrl-O D	Ctrl-K Y	delete block
Ctrl-O W	Ctrl-K W	write block to file
Ctrl-O J B	Ctrl-Q B	move to beginning of block
Ctrl-O J E	Ctrl-Q K	move to end of block
		Search & replace commands
Ctrl-Q F		search for string
Ctrl-Q A		search string and replace it
Ctrl-L		repeat last find or replace command
Ctrl-Q L		find & replace all remaining
Ctrl-Q V		move to last find or replace
		List structure commands
Ctrl-^ D	Alt-D	move forward a list
Ctrl-^ S	Alt-S	move backward a list
Ctrl-^ C	Alt-C	move forward to next definition
Ctrl-^ R	Alt-R	move to beginning of definition
Ctrl-^ T	Alt-T	delete list beginning at cursor
Ctrl-^ !	Alt-!	evaluate list beginning at cursor
		File I/O commands
Ctrl-K D	Ctrl-K S	save text in a file
Ctrl-K A	Ctrl-K Q	abandon text and clear edit buffer
Ctrl-K R		merge a file at cursor

Appendix G. EMACS-like Editor Commands

This table summarizes the edit mode commands available when the **muLISP** editor is EMACS-like. The commands are divided into the same categories used in Section 7.1.

Control key	PC key	Alt key	Cursor control commands
Ctrl-F	→		move forward a character
Ctrl-B	←		move backward a character
Ctrl-X F	Ctrl →	Alt-F	move forward a word
Ctrl-X B	Ctrl ←	Alt-B	move backward a word
Ctrl-N	↓		move down a line
Ctrl-P	↑		move up a line
Ctrl-E	End		move to right end of line
Ctrl-A	Home		move to left end of line
	Ctrl-End		move to bottom row of screen
	Ctrl-Home		move to top row of screen
Ctrl-I	Tab		move right to next tab stop
	Shift-Tab		move left to previous tab stop
Ctrl-M	Enter		move to beginning of next line
Ctrl-J	Ctrl-Enter		move down line and indent
			Display control commands
Ctrl-V	PgDn		scroll text up a screen
Ctrl-X V	PgUp	Alt-V	scroll text down a screen
Ctrl-X >	Ctrl-PgDn		move to end of text
Ctrl-X <	Ctrl-PgUp		move to start of text
			Entering text commands
Ctrl-X I		Alt-I	switch text entry mode
Ctrl-O	Ins		insert a new line
Ctrl-Q			escape character
			Deleting text commands
Ctrl-D	Del		delete character under cursor
Ctrl-H	Backspace		delete character left of cursor
Ctrl-X D		Alt-D	delete word under cursor
Rub	Ctrl-Backspace		delete word left of cursor
Ctrl-K			delete right end of line
Ctrl-X K		Alt-K	delete left end of line
Ctrl-Y			restore last deleted line

Control key	PC key	Alt key	Block commands
Ctrl-U B	Ctrl-@	Ctrl-2	mark beginning of block
Ctrl-U E			mark end of block
Ctrl-U M			move block to cursor
Ctrl-U C			copy block at cursor
Ctrl-U D	Ctrl-W		delete block
Ctrl-X W		Alt-W	write block to file
			Search & replace commands
Ctrl-S			search for string
Ctrl-X %		Alt-%	search and replace string
Ctrl-L			find/replace again
Ctrl-X L			find/replace all remaining
			List structure commands
Ctrl-Z F	Ctrl-Z)		move forward an S-expr
Ctrl-Z B	Ctrl-Z (		move backward an S-expr
Ctrl-Z]			move forward to next defn
Ctrl-Z [			move to beginning of defn
Ctrl-Z K			delete S-expr
Ctrl-Z E	Ctrl-Z !	Alt-!	evaluate S-expr
			File I/O commands
Ctrl-X S		Alt-S	save file
Ctrl-X A		Alt-A	quit file & clear edit buffer
Ctrl-X R		Alt-R	merge file

Appendix H. Base Page

This is a listing of the **muLISP** base page. It contains several system constants and tables. The first two columns in each row of the listing are the hexadecimal and decimal offsets in the code segment of the constants and tables.

The following **Special Form Jump Table** provides space for up to 10 pairs of pointers to user-defined special form routines. The first entry of each pair points to the routine that handles pseudo-code arguments. The second entry points to the routine that handles *bona fide* data object arguments. See Chapter 9 for details on implementing user-defined special forms.

```
0180    384     DW      0,0             ;Space for pointers to a pair of routines
0184    388     DW      0,0
0188    392     DW      0,0
018C    396     DW      0,0
0190    400     DW      0,0
0194    404     DW      0,0
0198    408     DW      0,0
019C    412     DW      0,0
01A0    416     DW      0,0
01A4    420     DW      0,0
```

The following **Default Read Table** specifies the default character type for the 256 different ASCII characters. The bits in each byte in the table describe the type of the corresponding ASCII character as follows:

Bit	Mask	Character type	Bit	Mask	Character type
0	01H	whitespace	4	10H	interrupt
1	02H	single escape	5	20H	terminating macro
2	04H	multiple escape	6	40H	nonterminating macro
3	08H	break	7	80H	comment macro

```
;                       @ A B C D E F G H I J K L M N O
0200    512     DB      0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0

;                       P Q R S T U V W X Y Z [ \ ] ^ _
0210    528     DB      0,0,0,0,0,0,0,0,0,0,0,0,16,0,0,0,0

;                         ! " # $ % & ' ( ) * + , - . /
0220    544     DB      1,8,40,8,8,8,8,40,40,40,8,8,40,8,8,8

;                       0 1 2 3 4 5 6 7 8 9 : ; < = > ?
0230    560     DB      0,0,0,0,0,0,0,0,0,0,8,128,8,8,8,8

;                       @ A B C D E F G H I J K L M N O
0240    576     DB      8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;                       P Q R S T U V W X Y Z [ \ ] ^ _
0250    592     DB      0,0,0,0,0,0,0,0,0,0,0,8,10,40,8,8
```

```
;                                   ` a b c d e f g h i j k l m n o
0260    608     DB      8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;                                   p q r s t u v w x y z { | } ~ RUB
0270    624     DB      0,0,0,0,0,0,0,0,0,0,0,8,12,8,8,0

0280    640     DB      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

0290    656     DB      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02A0    672     DB      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02B0    688     DB      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02C0    704     DB      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02D0    720     DB      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02E0    736     DB      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02F0    752     DB      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

The following **Default Macro Character Function Table** contains pointers to the default definitions for macro characters. The first word of the table is the number of entries in the table. There is room for up to 10 entries. An entry consists of a byte containing the ASCII code for a character followed by a word pointing to its macro definition; odd pointers imply machine language routines, even imply pseudo-code.

```
0300    768     DW      4           ;Entries in table
0302    770     DB      "'"         ;Single quote mark
                DW      RDQUOTE     ;Single quote mark macro function pointer
0305    773     DB      '('         ;Left parenthesis
                DW      RDLIST      ;Left parenthesis macro function pointer
0308    776     DB      ';'         ;Semicolon
                DW      RDCOMMENT   ;Semicolon macro function pointer
030B    779     DB      '"'         ;Double quote mark
                DW      RDSTRING    ;Double quote mark macro function pointer
030E    782     DB      0           ;Space for a macro character
                DW      0           ;Space for macro function pointer
0311    785     DB      0
                DW      0
0314    788     DB      0
                DW      0
0317    791     DB      0
                DW      0
031A    794     DB      0
                DW      0
031D    797     DB      0
                DW      0
```

The following words are the **pseudo-registers** used by REGISTER and INTERRUPT (see Section 5.8.7).

```
0320    800     DW      0       ;Register 0 (AX)
0322    802     DW      0       ;Register 1 (BX)
0324    804     DW      0       ;Register 2 (CX)
0326    806     DW      0       ;Register 3 (DX)
0328    808     DW      0       ;Register 4 (SI)
032A    810     DW      0       ;Register 5 (DI)
032C    812     DW      0       ;Register 6 (BP)
032E    814     DW      0       ;Register 7 (DS)
0330    816     DW      0       ;Register 8 (ES)
0332    818     DW      0       ;Register 9 (FLAGS)
```

The following words contain the minimum number of bytes that must be free in the atom, cons, and stack spaces after a garbage collection and reallocation. If any space has less than the required minimum, a Memory Full error break occurs.

```
0334    820     DW      200H    ;Minimum atom space bytes
0336    822     DW      200H    ;Minimum cons space bytes
0338    824     DW      400H    ;Minimum stack space bytes
```

After a garbage collection, the data space reallocator sizes the stack space so that the following word is the ratio of the total free space to the free stack space.

```
033A    826     DW      32      ;Ratio of the total to the stack space
```

During initialization, **muLISP** changes the 8086 code segment register to that returned by the call (CSMEMORY) (see Section 5.8.7), moves the base page from the original code segment to the new code segment, and stores the paragraph address of the original code segment in the following word:

```
0342    834     DW      ?       ;Original code segment
```

A data space reallocation occurs after a garbage collection if, for any data space, the ratio of the size of the space to the optimum size of the space is greater than [0344H]/[0346H] or less than [0346H]/[0344H].

```
0344    836     DW      4       ;Reallocation sensitivity
0346    838     DW      3       ;Reallocation sensitivity
```

The following word is the paragraph address above which memory will *not* be used. If this word is zero, all available memory is used, up to a maximum of 640K.

```
0348    840     DW      0       ;Maximum paragraph address
```

The following word is the default file and printer line length.

```
034A    842     DW      79      ;Default line length
```

The following word is used by TONE to initialize a counter which determines the duration of tones and pauses (see Section 5.8.7). If the word is zero when **muLISP** starts, the word is automatically calibrated using the system clock.

```
034C    844     DW      0               ;Timing counter word
```

The following byte is the mask used to strip the parity bit from characters read from files. Make this byte 0FFH (255) to prevent stripping any bits.

```
034E    846     DB      7FH             ;Parity bit mask for file input
```

The following byte is the character sent to the console to sound the console bell. Make this byte 0 to deactivate the bell.

```
034F    847     DB      7               ;Console bell character
```

The following bytes are the characters used by PRIN1 to delimit special characters when the control variable *PRINT-ESCAPE* is nonNIL.

```
0350    848     DB      '\'             ;Single escape characters
0351    849     DB      '|'             ;Multiple escape characters
```

The following byte controls the frequency at which the console is checked for a console interrupt character. Value can range from 1 for most frequent to 255 for least.

```
0352    850     DB      100             ;Keyboard interrupt sensitivity
```

If the following byte is nonzero, whitespace characters that terminate a call to READ will be unread so that they can be reread by the next call to READ. If the byte is zero, whitespace characters are not unread.

```
0353    851     DB      -1              ;Read preserving white space flag
```

If the EOF flag byte is nonzero, an End-Of-File error break occurs when the EOF character is encountered while reading a file. If the EOF flag byte is zero, an End-Of-File error break occurs only when an attempt is made to read past the *physical* end of a file. The EOF character byte is the character used to mark the *logical* end of a file.

```
0354    852     DB      -1              ;Logical EOF flag
0355    853     DB      1AH             ;EOF character
```

The following byte is the exit code returned by **muLISP** if the function SYSTEM is called without arguments (see Section 5.6.4) or if the **System** break option is chosen (see Section 5.10.3).

```
0356    854     DB      0               ;Default exit code
```

The following byte stores the computer type number used by the console screen function (see Section 5.8.6). This table lists the number associated with each computer type:

0 = Unknown computer type
1 = Generic MS-DOS computer
2 = IBM PC or "look-alike"
3 = ANSI or VT-100 Terminal
4 = TI Professional Computer
5 = Zenith Z-100 or VT-52
6 = Hewlett-Packard HP-150
7 = Hewlett-Packard HP-110
8 = NEC APC or ADM-3A
9 = NEC PC-9801
10 = Fujitsu Computer

```
0357    855     DB      0       ;Computer type byte
```

The following byte stores the printer type number used by the PRINT-SCREEN function. (PRINT-SCREEN *row column*) sends a text or graphics image, depending on the current video mode, of the current muLISP window to the system printer. Set the printer type byte to 0 for generic non-graphics printers, 1 for HP LaserJet compatible printers, 2 for Epson compatible printers, and 3 for IBM Proprinter compatible printers.

```
0358    856     DB      0       ;Printer type byte
```

When in a graphics video mode (see Appendix A), **muLISP** must provide its own blinking cursor. The following word stores the period used for the cursor in 60 millisecond units.

```
035A    858     DW      5       ;Graphics mode cursor period
```

If the following byte is zero, the pseudo-code compiler is disabled in COM files created by the function SAVE. If this byte is nonzero, the pseudo-code compiler is not disabled. *Only COM files with the pseudo-code compiler disabled can be sold as Runtime Systems* (see Section 5.9.4).

```
035C    860     DB      0       ;Preserve pseudo-code compiler flag
```

The following string is the read-eval-print loop prompt (see Section 5.10.2). It can consist of a maximum of five ASCII characters, and it must be followed by a 0 byte.

```
035D    861     DB      '$ ',0  ;muLISP prompt string
```

The following string is the default source file extension used by the function LOAD.

```
0363    867     DB      'LSP'   ;Default source file name extension
```

If running on a PC compatible computer and the following byte is nonzero, the cursor temporarily becomes a full block during garbage collections and data space reallocations.

```
0366    870     DB      -1      ;GC cursor indication flag
```

If the following byte is nonzero, input of a character whose high order bit is set is recognized as the first byte of a two-byte KANJI character.

```
0367    871     DB      0           ;Recognize KANJI characters flag
```

If the following byte is zero, the header message is *not* displayed when **muLISP** starts up (see Section 2.1).

```
0368    872     DB      -1          ;Display header message
```

The following character is used by the functions PLOT-DOT, PLOT-LINE, and PLOT-CIRCLE for plotting points when in a text video mode (see Section 5.8.6).

```
0369    873     DB      '.'         ;Character mode graphics character
```

The following characters are used by the line-editor to indicate that more text is to the right and/or left of the displayed part of the line.

```
036A    874     DB      '<'         ;More left indicator character
036B    875     DB      '>'         ;More right indicator character
```

If running on a PC compatible computer in a graphics mode having a character height of 8 pixels (see MAKE-WINDOW in Section 5.8.6) and the following byte is 1, an internal 8 by 8 character generator is used to display ASCII characters 128 through 255. This 1K character generator table, at offset 400H to 7FFH in the original code segment, can be modified as desired. If the following byte is 0, the default external character generator is used.

```
036C    876     DB      1           ;Use internal 8 by 8 ASCII character generator
```

If one or more of the following words are nonzero, the size of the **muLISP** console window (see Section 5.21) is limited accordingly.

```
036E    878     DW      0           ;Limit on base row
0370    880     DW      0           ;Limit on base column
0372    882     DW      0           ;Limit on rows
0374    884     DW      0           ;Limit on columns
```

When OPEN-OUTPUT-FILE creates a new file or DIRECTORY finds a list of files, the following word stores the file attribute used when making the required MS-DOS system calls (see an MS-DOS system programmers manual for details).

```
0376    886     DW      0           ;MS-DOS file attribute
```

The following **Line Editor Jump Table** contains the address of the machine language routine called when a control key or extended function key is pressed while using the **muLISP** line editor (see Section 5.10.5).

```
0380    896     DW      RHTCHR      ;Index 1:  Ctrl-D or -->
0382    898     DW      LFTCHR      ;Index 2:  Ctrl-S or <--
0384    900     DW      RHTWRD      ;Index 3:  Ctrl-F or Ctrl -->
0386    902     DW      LFTWRD      ;Index 4:  Ctrl-A or Ctrl <--
0388    904     DW      RHTEND      ;Index 5:  Ctrl-\ or End
038A    906     DW      LFTEND      ;Index 6:  Ctrl-] or Home
038C    908     DW      RHTTAB      ;Index 7:  Ctrl-I or Tab

038E    910     DW      DELCHR      ;Index 8:  Ctrl-G or Del
0390    912     DW      DELWRD      ;Index 9:  Ctrl-T
0392    914     DW      DELLFT      ;Index 10: Ctrl-H, Rubout, or Backspace
0394    916     DW      DELLIN      ;Index 11: Ctrl-Y

0396    918     DW      ESCMOD      ;Index 12: Ctrl-P
0398    920     DW      CMDCHD      ;Index 13: Ctrl-Q
039A    922     DW      UNDLIN      ;Index 14: Ctrl-U or F3
039C    924     DW      INSTOG      ;Index 15: Ctrl-V or Ins
```

To change the above control and extended function key correspondence, use the KEY-TRANSLATION function. If *char* and *index* are nonnegative integers and *char* < 256 and *index* < 16, (KEY-TRANSLATION *char index*) sets the routine called for ASCII character *char* to *index* and returns the previous index. If *char* is between 256 and 511 inclusive, the extended function character key corresponding to *char*-256 (see Appendix D) is set instead. (KEY-TRANSLATION *char*) returns the current translation index of *char*. If the index of a character is 0 and the character is pressed while using the line editor, the edit is terminated and *LINE-TERMINATOR* is set to the control character's byte value. If the index of an extended function key is 0 and the key is pressed, the edit is terminated and *LINE-TERMINATOR* is set to minus the key's character code.

When *GCSTAT* is nonNIL, the free memory in each data space is displayed before and after garbage collections (see Section 5.7.6). The data for the last garbage collection is stored in the following table:

```
03C6    966     DB      0,0,0       ;Pre-GC free atom space bytes
03C9    969     DB      0,0,0       ;Pre-GC free cons space bytes
03CC    972     DB      0,0,0       ;Pre-GC free string space bytes
03CF    975     DB      0,0,0       ;Pre-GC free vector space bytes
03D2    978     DB      0,0,0       ;Pre-GC free pseudo-code space bytes
03D5    981     DB      0,0,0       ;Pre-GC free stack space bytes
03D8    984     DB      0,0,0       ;Pre-GC free total space bytes

03DB    987     DB      0,0,0       ;Post-GC free atom space bytes
03DE    990     DB      0,0,0       ;Post-GC free cons space bytes
03E1    993     DB      0,0,0       ;Post-GC free string space bytes
03E4    996     DB      0,0,0       ;Post-GC free vector space bytes
03E7    999     DB      0,0,0       ;Post-GC free pseudo-code space bytes
03EA    1002    DB      0,0,0       ;Post-GC free stack space bytes
03ED    1005    DB      0,0,0       ;Post-GC free total space bytes
```

Appendix I. Global Storage Area

This is a listing of the **muLISP** global storage area. It shows the offset address of the variables that delimit the data spaces and other system variables. The first two columns in each row of the listing are the hexadecimal and decimal offsets in the data segment of the variables.

```
                                                ;Object space variables
                BASSYM  EQU             100H    ;Base of symbols
0000    0       ENDSYM  DW              ?       ;End of symbols
0002    2       ENDNUM  DW              ?       ;End of numbers
0004    4       ENDATM  DW              ?       ;End of atom space
0006    6       ENDCNS  DW              ?       ;End of conses
0008    8       MAXCNS  DW              ?       ;Max size of cons space

                                                ;Segment pointer variables
0010    16      DEFSEG  DW              ?       ;Function/length cell seg
0012    18      PNPSEG  DW              ?       ;Print name/vector cell seg
0014    20      PNSSEG  DW              ?       ;Print name string segment
0016    22      VECSEG  DW              ?       ;Number vector segment

                                                ;Stack segment variables
                BASSTK  EQU             400H    ;Base of stack space
0022    34      BASCOD  DW              ?       ;End of stk/base of code space
0024    36      ENDCOD  DW              ?       ;End of code space
0026    38      MAXCOD  DW              ?       ;Maximum size of code space

                                                ;Pname segment variables
0028    40      ENDSTR  DW              ?       ;End of print name strings
002A    42      MAXSTR  DW              ?       ;Max size of string space

                                                ;Vector segment variables
002C    44      ENDVEC  DW              ?       ;End of number vectors
002E    46      MAXVEC  DW              ?       ;Max size of vector space

                                                ;File control block variables
0030    48      BASFCB  DW              ?       ;Base of FCB space in CS
0032    50      IFCB    DW              ?       ;SIF FCB pointer
0034    52      OFCB    DW              ?       ;SOF FCB pointer
0036    54      ENDFCB  DW              ?       ;End of FCB space in CS

0038    56      THRVAL  DW              ?       ;Thrown value (0 = inactive)
003A    58      PRECSN  DW              ?       ;Current precision
003C    60      UNDFLO  DW              ?       ;Current underflow

003E    62      GCCTR   DW              ?       ;Garbage collection counter
0040    64      RACTR   DW              ?       ;Reallocation counter
0042    66      ORICS   DW              ?       ;Original code segment
```

Appendix J. Assembly Language Routine

This is an annotated assembly language listing of the COPY routine discussed in Section 9.4:

```
        ASSUME   CS:CSEG
CSEG    SEGMENT

ENDATM  EQU      04H                    ;Offset of ENDATM in DS
CONS    EQU      0C41H                  ;Offset of CONS returned by (GETD 'CONS)

                                        ;FUNCTION:  (COPY OBJ)
COPY:   JCXZ     RETNIL                 ;Return NIL if no arguments
        MOV      SI,[BP]                ;SI: OBJ
COPYSI: CMP      SI,WORD PTR DS:[ENDATM]
        JB       COPY1                  ;Jump if (ATOM OBJ)
        MOV      [BP],SI                ;Push OBJ on the variable stack
        INC      BP
        INC      BP
        MOV      SI,[SI]                ;SI: (CAR OBJ)
        EVEN
        CALL     COPYSI                 ;DI: (COPY (CAR OBJ))
        XCHG     DI,[BP-2]              ;Exchange OBJ with stack top
        MOV      SI,ES:[DI]             ;SI: (CDR OBJ)
        EVEN
        CALL     COPYSI                 ;DI: (COPY (CDR OBJ))
        DEC      BP                     ;Restore BP
        DEC      BP
        MOV      [BP+2],DI              ;Store DI on top of the variable stack
        MOV      CX,4                   ;Set CX to indicate 2 arguments
        MOV      AX,OFFSET CONS
        JMP      AX                     ;DI: Cons copies of CAR & CDR

COPY1:  MOV      DI,SI                  ;DI: OBJ
        RET

RETNIL: MOV      DI,100H                ;DI: NIL
        RET

CSEG    ENDS
        END      COPY
```

Function Index

This is an alphabetical list of the functions, special forms, macros, variables, constants, options, and keywords in **muLISP**. The brackets following functions and special forms enclose a list summarizing their arguments. The next column gives the category of primitively defined functions or the source file defining the nonprimitive functions. The remaining columns give the type of functions and the page on which they are described.

Name [arguments]	Category	Type	Page
ABS [*n*]	Numerical	Function	138
ABSTRAP [*object*]	Break	Function	277
ACONS [*key, object, a-list*]	A-list	Function	94
ACOS [*n*]	IRRATNAL.LSP	Function	291
ACOSH [*n*]	IRRATNAL.LSP	Function	293
ACOT [*n, m*]	IRRATNAL.LSP	Function	291
ACOTH [*n*]	IRRATNAL.LSP	Function	293
ACSC [*n*]	IRRATNAL.LSP	Function	291
ACSCH [*n*]	IRRATNAL.LSP	Function	293
ADD1 [*n*]	Numerical	Function	133
ADJOIN [*object, list, test*]	Constructor	Function	61
ALLOCATE [*n*]	Memory	Function	204
ALPHA-CHAR-P [*symbol*]	Character	Function	152
ALPHANUMERICP [*symbol*]	Character	Function	152
AND [$form_1$, ..., $form_n$]	Logical	Special form	51
APPEND [$list_1$, ..., $list_n$]	Constructor	Function	63
APPLY [*function*, arg_1, ..., arg_n, *arglist*]	Application	Function	190
AREF [*array*, $subscript_1$, ...]	ARRAY.LSP	Macro	283
ARRAY-DIMENSION [*array, axis*]	ARRAY.LSP	Function	284
ARRAY-DIMENSIONS [*array*]	ARRAY.LSP	Function	284
ARRAY-IN-BOUNDS [*array*, $subscript_1$, ...]	ARRAY.LSP	Function	285
ARRAY-RANK [*array*]	ARRAY.LSP	Function	284
ARRAY-TOTAL-SIZE [*array*]	ARRAY.LSP	Function	284
ARRAYP [*object*]	ARRAY.LSP	Function	285
ASCII [*atom*]	Character	Function	156
ASEC [*n*]	IRRATNAL.LSP	Function	291
ASECH [*n*]	IRRATNAL.LSP	Function	293
ASH [*n, count*]	Numerical	Function	146
ASIN [*n*]	IRRATNAL.LSP	Function	291
ASINH [*n*]	IRRATNAL.LSP	Function	293

ASSOC [*key, a-list, test*]	A-list	Function	95
ASSOC-IF [*test, a-list*]	A-list	Function	95
ATAN [*n, m*]	IRRATNAL.LSP	Function	291
ATANH [*n*]	IRRATNAL.LSP	Function	293
ATOM [*object*]	Recognizer	Function	42
AUTO-NEWLINE: T	Output	Control var.	225
BACK [*distance*]	GRAPHICS.LSP	Function	311
BACKGROUND-COLOR [*n*]	Screen	Function	234
BENCH-MARK [*form*]	Hardware	Macro	238
BINARY-LOAD [*file, offset*]	Hardware	Function	247
BLANK-COMPRESS: NIL	Output	Control var.	225
BORDER-COLOR [*n*]	Screen	Function	234
BOUNDP [*symbol*]	Symbol	Function	101
BOUNDP-IN-CLOSURE [*closure, symbol*]	CLOSURE.LSP	Function	308
BREAK [*object, message*]	Break	Function	273
BREAK	Break	Variable	273
BREAK-CHARS [*list, flag*]	Input	Function	208
BUTLAST [*list, n*]	Constructor	Function	66
BUTTON-PRESSES [*button*]	MOUSE.LSP	Function	315
BUTTON-RELEASES [*button*]	MOUSE.LSP	Function	315
CAAAAR [*object*]	Selector	Function	54
CAAADR [*object*]	Selector	Function	54
CAAAR [*object*]	Selector	Function	54
CAADAR [*object*]	Selector	Function	54
CAADDR [*object*]	Selector	Function	54
CAADR [*object*]	Selector	Function	54
CAAR [*object*]	Selector	Function	54
CADAAR [*object*]	Selector	Function	54
CADADR [*object*]	Selector	Function	54
CADAR [*object*]	Selector	Function	54
CADDAR [*object*]	Selector	Function	54
CADDDR [*object*]	Selector	Function	54
CADDR [*object*]	Selector	Function	54
CADR [*object*]	Selector	Function	54
CAR [*object*]	Selector	Function	53
CASE [*keyform*, $case_1$, ..., $case_n$]	Control	Macro	175
CATCH [*tag*, $form_1$, ..., $form_n$]	Control	Special form	179

CDAAAR [*object*]	Selector	Function	54
CDAADR [*object*]	Selector	Function	54
CDAAR [*object*]	Selector	Function	54
CDADAR [*object*]	Selector	Function	54
CDADDR [*object*]	Selector	Function	54
CDADR [*object*]	Selector	Function	54
CDAR [*object*]	Selector	Function	54
CDDAAR [*object*]	Selector	Function	54
CDDADR [*object*]	Selector	Function	54
CDDAR [*object*]	Selector	Function	54
CDDDAR [*object*]	Selector	Function	54
CDDDDR [*object*]	Selector	Function	54
CDDDR [*object*]	Selector	Function	54
CDDR [*object*]	Selector	Function	54
CDR [*object*]	Selector	Function	53
CEILING [*n*, *m*]	Numerical	Function	140
CHANGE-WINDOW [ ]	WINDOWS.LSP	Function	341
CHAR [*atom*, *n*]	String	Function	160
CHAR-CODE [*symbol*]	Character	Function	156
CHAR-DOWNCASE [*symbol*]	Character	Function	157
CHAR-EQUAL [sym_1, ..., sym_n]	Character	Function	155
CHAR-GREATERP [sym_1, ..., sym_n]	Character	Function	155
CHAR-LESSP [sym_1, ..., sym_n]	Character	Function	155
CHAR-NOT-EQUAL [sym_1, ..., sym_n]	Character	Function	155
CHAR-NOT-GREATERP [sym_1, ..., sym_n]	Character	Function	155
CHAR-NOT-LESSP [sym_1, ..., sym_n]	Character	Function	155
CHAR-UPCASE [*symbol*]	Character	Function	157
CHAR= [$symbol_1$, ..., $symbol_n$]	Character	Function	154
CHAR< [$symbol_1$, ..., $symbol_n$]	Character	Function	154
CHAR> [$symbol_1$, ..., $symbol_n$]	Character	Function	154
CHAR<= [$symbol_1$, ..., $symbol_n$]	Character	Function	154
CHAR>= [$symbol_1$, ..., $symbol_n$]	Character	Function	154
CHAR/= [$symbol_1$, ..., $symbol_n$]	Character	Function	154
CLEAR-INPUT [*source*]	Input	Function	211
CLEAR-SCREEN [ ]	Screen	Function	232
CLOSE-INPUT-FILE [*file*]	Input file	Function	251
CLOSE-OUTPUT-FILE [*file*]	Output file	Function	257
CLOSE-WINDOW	WINDOWS.LSP	Keyword	340
CLOSURE [*varlist*, *function*]	CLOSURE.LSP	Function	304

CLOSURE-ALIST [*closure*]	CLOSURE.LSP	Function	308
CLOSURE-FUNCTION [*closure*]	CLOSURE.LSP	Function	308
CLOSURE-VARIABLES [*closure*]	CLOSURE.LSP	Function	308
CLOSUREP [*closure*]	CLOSURE.LSP	Function	306
CLR [ ]	GRAPHICS.LSP	Function	310
CODE-CHAR [*n*]	Character	Function	156
COLUMN [ ]	Screen	Function	231
COLUMNS [ ]	Screen	Function	231
COMMENT [*comments*]	Evaluation	Macro	188
COMMENT	Input	Keyword	215
COMPILE [*name*]	COMPILE.LSP	Function	346
COMPILE-FILE [*compile-files*, *load-files*]	COMPILE.LSP	Function	348
:CONC-NAME [*prefix*]	STRUCTUR.LSP	Option	295
COND [$cond_1$, ..., $cond_n$]	Control	Special form	174
CONDENSE: NIL	Definition	Control var.	120
CONS [obj_1, obj_2]	Constructor	Function	60
CONSOLE-INTERRUPT [ ]	Break	Function	276
CONSP [*object*]	Recognizer	Function	43
CONSTANTP [*object*]	Evaluation	Function	187
:CONSTRUCTOR [*name*, *arglist*]	STRUCTUR.LSP	Option	295
CONTINUE-PROMPT [ ]	WINDOWS.LSP	Function	341
:COPIER [*name*]	STRUCTUR.LSP	Option	295
COPY-ALIST [*a-list*]	A-list	Function	94
COPY-CHAR-TYPE [$char_1$, $char_2$, *flag*]	Input	Function	214
COPY-CLOSURE [*closure*]	CLOSURE.LSP	Function	306
COPY-LIST [*list*]	Constructor	Function	64
COPY-TREE [*object*]	Constructor	Function	64
COS [*n*]	IRRATNAL.LSP	Function	290
COS-DEG [*angle*]	GRAPHICS.LSP	Function	309
COSH [*n*]	IRRATNAL.LSP	Function	292
COT [*n*]	IRRATNAL.LSP	Function	290
COTH [*n*]	IRRATNAL.LSP	Function	292
COUNT [*object*, *list*, *test*]	Search	Function	82
COUNT-IF [*test*, *list*]	Search	Function	82
CREATE-WINDOW	WINDOWS.LSP	Keyword	340
CSC [*n*]	IRRATNAL.LSP	Function	290
CSCH [*n*]	IRRATNAL.LSP	Function	292
CSMEMORY [*offset*, *value*, *flag*]	Hardware	Function	243
CURSOR-LINES [*start-line*, *end-line*]	Screen	Function	236

DECF [*place*, *n*]	Numerical	Macro	135
DECQ [*symbol*, *n*]	Numerical	Special form	134
DEFAULT-DRIVE [*drive*]	Directory	Function	264
DEFAULT-PATH [*drive*, *path*]	Directory	Function	264
DEFFLAVOR [*name*, *vars*, *flavors*, opt_1, ...]	FLAVORS.LSP	Macro	298
DEFMACRO [*sym*, *arglist*, $form_1$, ...]	Definition	Special form	119
DEFMETHOD [(*flv type meth*), *args*, $form_1$, ...]	FLAVORS.LSP	Macro	300
DEFSTRUCT [(*name opt1* ...), $slot_1$, ...]	STRUCTUR.LSP	Macro	294
DEFUN [*symbol*, *arglist*, $form_1$, ..., $form_n$]	Definition	Special form	118
DEFUN [*symbol*, (*ftype arglist* $form_1$ ... $form_n$)]	Definition	Special form	118
DEFUN [*symbol*, *address*]	Definition	Special form	118
DEFWHOPPER [(*flavor meth*), *args*, $form_1$, ...]	FLAVORS.LSP	Macro	301
DELETE [*item*, *list*, *test*]	Deletion	Function	77
DELETE-COMPILER [ ]	COMPILE.LSP	Macro	347
DELETE-DUPLICATES [*list*, *test*]	Deletion	Function	79
DELETE-FILE [*file*]	File	Function	262
DELETE-IF [*test*, *list*]	Deletion	Function	78
DELETE-LINES [*n*]	Screen	Function	232
DELETE-NTH [*list*, *n*]	Modifier	Function	75
DENOMINATOR [*n*]	Numerical	Function	139
DENTRAP [*object*]	Break	Function	277
DIGIT-CHAR-P [*symbol*, *radix*]	Character	Function	153
DIRECTORY [*path*]	Directory	Function	264
DISPLAY-PAGE [*n*, *active-only*]	Screen	Function	235
DIVIDE [*n*, *m*]	Numerical	Function	143
DO [*letlist*, $form_1$, ..., $form_n$]	Control	Macro	177
DO* [*letlist*, $form_1$, ..., $form_n$]	Control	Macro	177
DOLIST [(*var list result*), $form_1$, ..., $form_n$]	Control	Macro	178
DOS [*command*]	Control	Function	183
DOT [*x*, *y*]	GRAPHICS.LSP	Function	309
DOTIMES [(*var count result*), $form_1$, ..., $form_n$]	Control	Macro	178
DRIVER [*file*]	Driver	Function	271
DRIVER: 'DRIVER	Driver	Control var.	270
DSMEMORY [*offset*, *value*, *flag*]	Hardware	Function	243
EIGHTH [*list*]	Selector	Function	56
ENCODE-UNIVERSAL-TIME [*s*, *m*, *h*, *d*, *mo*, *y*, *z*]	Hardware	Function	239
ENDP [*object*]	Recognizer	Function	44
EQ [obj_1, obj_2]	Comparator	Function	45

EQL [obj_1, obj_2]	Comparator	Function	46
EQUAL [obj_1, obj_2, *test*]	Comparator	Function	47
ERROR-BEEP [ ]	WINDOWS.LSP	Function	341
EVAL [*form*]	Evaluation	Function	184
EVAL-FUNCTION-P [*symbol*]	Definition	Function	114
EVENP [*object*]	Numerical	Function	126
EVERY [*test*, $list_1$, ..., $list_n$]	Mapping	Function	195
EXACT-MODE: NIL	Numerical	Control var.	122
EXACT-MODEP [ ]	IRRATNAL.LSP	Function	286
EXECUTE [*program*, *command*]	Control	Function	183
EXECUTE-OPTION [*prompt*, *option-tree*]	WINDOWS.LSP	Function	341
EXP [*n*]	IRRATNAL.LSP	Function	286
EXPT [*n*, *m*]	IRRATNAL.LSP	Function	287
FBOUNDP [*symbol*]	Definition	Function	114
FIFTH [*list*]	Selector	Function	56
FILE-DATE [*file*]	File	Function	263
FILE-LENGTH [*file*]	File	Function	263
FILE-LINE-LENGTH [*n*, *file*]	Output file	Function	261
FILE-READ-POSITION [*n*, *file*]	Input file	Function	254
FILE-WRITE-POSITION [*n*, *file*]	Output file	Function	260
FILL [*list*, *object*]	Modifier	Function	71
FIND [*object*, *list*, *test*]	Search	Function	80
FIND-IF [*test*, *list*]	Search	Function	80
FINDSTRING [$atom_1$, $atom_2$, *n*]	String	Function	166
FIRST [*list*]	Selector	Function	56
FIRSTN [*n*, *list*]	Constructor	Function	66
FLAG [*symbol*, *flag*]	Flag	Function	111
FLAGP [*symbol*, *flag*]	Flag	Function	111
FLOOR [*n*, *m*]	Numerical	Function	140
FMAKUNBOUND [*symbol*]	Definition	Function	117
FOREGROUND-COLOR [*n*]	Screen	Function	234
FORWARD [*distance*]	GRAPHICS.LSP	Function	311
FOURTH [*list*]	Selector	Function	56
FREE-LIST: NIL	Memory	Variable	203
FRESH-LINE [*sink*]	Output	Function	223
FUNCALL [*function*, arg_1, ..., arg_n]	Application	Function	189
FUNCTIONP [*object*]	Evaluation	Function	187

GCD [n_1, ..., n_m]	Numerical	Function	136
GCDTRAP [obj_1, obj_2]	Break	Function	277
GCHOOK: NIL	Memory	Control var.	202
GCSTAT: NIL	Memory	Control var.	201
GENSYM [*atom*]	String	Function	168
GENSYM-COUNT: 0	String	Control var.	168
GENSYM-PREFIX: G	String	Control var.	168
GET [*symbol, key, default*]	Property	Function	109
GET-DECODED-TIME [ ]	Hardware	Function	239
GET-PROPERTIES [*plist, key-list*]	Property	Function	109
GET-MACRO-CHAR [*char, flag*]	Input	Function	215
GET-UNIVERSAL-TIME [ ]	Hardware	Function	239
GETD [*symbol, flag*]	Definition	Function	112
GETF [*plist, key, default*]	Property	Function	109
GETSET [*parameter*]	Directory	Function	265
:GETTABLE-INSTANCE-VARIABLES	FLAVORS.LSP	Option	298
GO-DOS [ ]	WINDOWS.LSP	Function	342
HIDE-MOUSE [ ]	MOUSE.LSP	Function	314
HISTORY-LENGTH: 25	DEBUG.LSP	Control var.	338
IDENTITY [*object*]	Evaluation	Function	186
IF [*testform, thenform, elseform*]	Control	Special form	173
IGNORE-CASE: NIL	Input	Control var.	219
INCF [*place, n*]	Numerical	Macro	135
:INCLUDE [*structure*]	STRUCTUR.LSP	Option	296
:INCLUDED-FLAVORS [$flavor_1$, ..., $flavor_n$]	FLAVORS.LSP	Option	299
INCQ [*symbol, n*]	Numerical	Special form	134
:INITABLE-INSTANCE-VARIABLES	FLAVORS.LSP	Option	298
:INITIAL-CONTENTS *expression*	ARRAY.LSP	Option	282
:INITIAL-ELEMENT *expression*	ARRAY.LSP	Option	282
INIT-WINDOW	WINDOWS.LSP	Control var.	339
INPUT-ECHO: NIL	Input	Control var.	218
INPUT-FILE [*file*]	Input file	Function	252
INPUT-FILE: NIL	Input file	Control var.	253
INPUT-FILE-P [*file*]	Input file	Function	252
INPUT-FILES [ ]	Input file	Function	252
INSERT-LINES [*n*]	Screen	Function	232
INSERT-MODE: T	Line editor	Control var.	280

INSERT-NTH [*item*, *list*, *n*]	Modifier	Function	75
INTEGER-LENGTH [*n*]	Numerical	Function	147
INTEGERP [*object*]	Recognizer	Function	41
INTERRUPT [*n*]	Hardware	Function	246
INTERRUPT-HOOK: 'CONSOLE-INTERRUPT	Break	Control var.	276
INTERSECTION [$list_1$, $list_2$, *test*]	Set	Function	90
ISQRT [*n*]	IRRATNAL.LSP	Function	287
LAMBDA	Definition	Keyword	112
LAST [*list*]	Selector	Function	57
LCM [n_1, ..., n_m]	Numerical	Function	137
LCMTRAP [obj_1, obj_2]	Break	Function	277
LCONC [*dotted-pair*, *list*]	Modifier	Function	76
LDIFF [*list*, *tail*]	Constructor	Function	67
LEFT [*angle*]	GRAPHICS.LSP	Function	310
LEFT-BUTTON	MOUSE.LSP	Variable	315
LENGTH [*object*]	Selector	Function	59
LET [*letlist*, $form_1$, ..., $form_n$]	Control	Macro	172
LET* [*letlist*, $form_1$, ..., $form_n$]	Control	Macro	172
LET-CLOSED [*letlist*, *function*]	CLOSURE.LSP	Macro	305
LINE [x_1, y_1, x_2, y_2]	GRAPHICS.LSP	Function	309
LINE-COLUMN	Line editor	Variable	279
LINE-EDIT [*text*, *point*, *col*, *width*]	Line editor	Function	279
LINE-POINT	Line editor	Variable	279
LINE-TERMINATOR	Line editor	Variable	279
LIST [obj_1, ..., obj_n]	Constructor	Function	62
LIST* [obj_1, ..., obj_n]	Constructor	Function	62
LIST-LENGTH [*object*]	Selector	Function	59
LISTEN [*source*]	Input	Function	211
LISTP [*object*]	Recognizer	Function	43
LN [*n*]	IRRATNAL.LSP	Function	288
LOAD [*file*, *display*]	Memory image	Function	268
LOAD-VERBOSE: T	Memory image	Control var.	268
LOCATE-IN-CLOSURE [*closure*, *symbol*]	CLOSURE.LSP	Function	307
LOCATION [*object*]	Hardware	Function	245
LOG [*n*, *base*]	IRRATNAL.LSP	Function	288
LOGAND [n_1, ..., n_m]	Numerical	Function	145
LOGBITP [*index*, *n*]	Numerical	Function	144
LOGEQV [n_1, ..., n_m]	Numerical	Function	145

LOGIOR [n_1, ..., n_m]	Numerical	Function	145
LOGNOT [n]	Numerical	Function	144
LOGTEST [n, m]	Numerical	Function	144
LOGXOR [n_1, ..., n_m]	Numerical	Function	145
LOOP [$form_1$, ..., $form_n$]	Control	Special form	176
LOWER-CASE-P [*symbol*]	Character	Function	151
MACRO	Definition	Keyword	112
MACRO-FUNCTION [*symbol*]	Macro	Function	197
MACRO-P [*symbol*]	Definition	Function	114
MACROEXPAND [*form*]	Macro	Function	198
MACROEXPAND: T	Macro	Control var.	199
MACROEXPAND-1 [*form*]	Macro	Function	198
MAKE-ARRAY [*dimensions*, opt_1, ..., opt_n]	ARRAY.LSP	Function	282
MAKE-INSTANCE [*flavor*, *vars*]	FLAVORS.LSP	Macro	302
MAKE-LIST [n, *object*, *list*]	Constructor	Function	64
MAKE-RANDOM-STATE [*state*]	Numerical	Function	149
MAKE-WINDOW [*row*, *column*, *rows*, *cols*]	Screen	Function	230
MAPC [*function*, $list_1$, ..., $list_n$]	Mapping	Function	191
MAPCAN [*function*, $list_1$, ..., $list_n$]	Mapping	Function	193
MAPCAR [*function*, $list_1$, ..., $list_n$]	Mapping	Function	192
MAPCON [*function*, $list_1$, ..., $list_n$]	Mapping	Function	193
MAPL [*function*, $list_1$, ..., $list_n$]	Mapping	Function	191
MAPLIST [*function*, $list_1$, ..., $list_n$]	Mapping	Function	192
MAX [n_1, ..., n_m]	Numerical	Function	130
MAXTRAP [obj_1, obj_2]	Break	Function	277
MEMBER [*object*, *list*, *test*]	Search	Function	84
MEMBER-IF [*test*, *list*]	Search	Function	84
MEMORY [*address*, *value*, *flag*]	Hardware	Function	242
MERGE [$list_1$, $list_2$, *test*]	Sorting	Function	99
:METHOD-COMBINATION	FLAVORS.LSP	Option	299
MIN [n_1, ..., n_m]	Numerical	Function	130
MINTRAP [obj_1, obj_2]	Break	Function	277
MINUSP [*object*]	Numerical	Function	125
MISMATCH [$list_1$, $list_2$, *test*]	Search	Function	83
MOD [n, m]	Numerical	Function	142
MOUSE-BOX [x_1, y_1, x_2, y_2]	MOUSE.LSP	Function	316
MOUSE-POSITION [x, y]	MOUSE.LSP	Function	314
MOUSE-RESET [ ]	MOUSE.LSP	Function	313

MOUSE-STATUS [ ]	MOUSE.LSP	Function	315
MOUSE-X	MOUSE.LSP	Function	314
MOUSE-Y	MOUSE.LSP	Function	314
MOVD [*symbol*$_1$, *symbol*$_2$]	Definition	Function	116
:NAMED	STRUCTUR.LSP	Option	296
NBUTLAST [*list*, *n*]	Modifier	Function	74
NCONC [*list*$_1$, ..., *list*$_n$]	Modifier	Function	72
NEQ [*obj*$_1$, *obj*$_2$]	Comparator	Function	45
NEQL [*obj*$_1$, *obj*$_2$]	Comparator	Function	46
NEW-CODE-SPACE [ ]	Memory	Function	205
NIL	Evaluation	Constant	44
NINTERSECTION [*list*$_1$, *list*$_2$, *test*]	Set	Function	90
NINTH [*list*]	Selector	Function	56
NLAMBDA	Definition	Keyword	112
NO-EVAL-FUNCTION-P [*symbol*]	Definition	Function	114
:NO-VANILLA-FLAVOR	FLAVORS.LSP	Option	299
NOT [*object*]	Logical	Function	50
NOTANY [*test*, *list*$_1$, ..., *list*$_n$]	Mapping	Function	194
NOTEVERY [*test*, *list*$_1$, ..., *list*$_n$]	Mapping	Function	195
NRECONC [*list*, *object*]	Modifier	Function	73
NREVERSE [*list*, *object*]	Modifier	Function	73
NSET-DIFFERENCE [*list*$_1$, *list*$_2$, *test*]	Set	Function	91
NSET-EXCLUSIVE-OR [*list*$_1$, *list*$_2$, *test*]	Set	Function	92
NSUBLIS [*a-list*, *object*, *test*]	A-list	Function	97
NSUBST [*new*, *old*, *object*, *test*]	Substitution	Function	87
NSUBST-IF [*new*, *test*, *object*]	Substitution	Function	88
NSUBSTITUTE [*new*, *old*, *list*, *test*]	Substitution	Function	85
NSUBSTITUTE-IF [*new*, *test*, *list*]	Substitution	Function	86
NTH [*n*, *list*]	Selector	Function	58
NTHCDR [*n*, *list*]	Selector	Function	58
NULL [*object*]	Recognizer	Function	44
NUMBERP [*object*]	Recognizer	Function	41
NUMERATOR [*n*]	Numerical	Function	139
NUMERIC-CHAR-P [*symbol*]	Character	Function	152
NUMTRAP [*object*]	Break	Function	277
NUNION [*list*$_1$, *list*$_2$, *test*]	Set	Function	89

OBLIST [ ]	Constructor	Function	69
ODDP [*object*]	Numerical	Function	126
OPEN-INPUT-FILE [*file*]	Input file	Function	250
OPEN-OUTPUT-FILE [*file, overwrite*]	Output file	Function	255
OR [$form_1$, ..., $form_n$]	Logical	Special form	52
ORDERP [obj_1, obj_2]	Comparator	Function	49
OTHERWISE	Control	Keyword	175
OUTPUT-ECHO: NIL	Output	Control var.	225
OUTPUT-FILE [*file*]	Output file	Function	258
OUTPUT-FILE: NIL	Output file	Control var.	259
OUTPUT-FILE-P [*file*]	Output file	Function	258
OUTPUT-FILES [ ]	Output file	Function	258
PACK [*list*]	String	Function	159
PACK* [$atom_1$, ..., $atom_n$]	String	Function	159
PAIRLIS [*keys, objects, a-list*]	A-list	Function	94
PALETTE-COLOR [*n*]	Screen	Function	234
PARSE-INTEGER [*string, radix*]	Input	Function	213
PEEK-CHAR [*type, source, eof-p, eof-v*]	Input	Function	210
PENDOWN [ ]	GRAPHICS.LSP	Function	310
PENUP [ ]	GRAPHICS.LSP	Function	310
PI [ ]	IRRATNAL.LSP	Function	289
PLOT-CIRCLE [*x, y, r, color*]	Screen	Function	237
PLOT-COLOR: 15	Screen	Control var.	237
PLOT-DOT [*x, y, color*]	Screen	Function	237
PLOT-LINE [x_1, y_1, x_2, y_2, *color*]	Screen	Function	237
PLUSP [*object*]	Numerical	Function	125
POP [*symbol*]	Symbol	Special form	106
PORTIO [*port, value, flag*]	Hardware	Function	241
POSITION [*object, list, test*]	Search	Function	81
POSITION-IF [*test, list*]	Search	Function	81
PPRINT [*object, sink*]	Output	Function	221
PRECISION [*n*]	Numerical	Function	122
:PREDICATE [*name*]	STRUCTUR.LSP	Option	295
PRIN1 [*object, sink*]	Output	Function	221
PRINC [*object, sink*]	Output	Function	221
PRINT [*object, sink*]	Output	Function	221
PRINT-BASE: 10	Output	Control var.	227
PRINT-DOWNCASE: NIL	Output	Control var.	226

PRINT-ESCAPE: T	Output	Control var.	226
PRINT-LENGTH [*atom*]	String	Function	167
PRINT-POINT: 7	Output	Control var.	228
PRINT-TRACE: NIL	DEBUG.LSP	Control var.	338
PRINTER-ECHO: NIL	Output	Control var.	225
PROBE-FILE [*file*]	File	Function	262
PROG1 [$form_1$, ..., $form_n$]	Control	Special form	171
PROG2 [$form_1$, ..., $form_n$]	Control	Macro	171
PROGN [$form_1$, ..., $form_n$]	Control	Special form	170
PROMPT-YN [*prompt*]	WINDOWS.LSP	Function	341
PSETQ [$symbol_1$, $form_1$, ..., $symbol_n$, $form_n$]	Symbol	Special form	104
PUSH [*form*, *symbol*]	Symbol	Special form	107
PUSHNEW [*form*, *symbol*, *test*]	Symbol	Macro	107
PUT [*symbol*, *key*, *object*]	Property	Function	108
PUTD [*symbol*, *definition*]	Definition	Function	115
QUIT-PROGRAM [ ]	WINDOWS.LSP	Function	342
QUOTE [*object*]	Evaluation	Special form	186
RANDOM [*n*, *state*]	Numerical	Function	148
RANDOM-STATE	Numerical	Control var.	149
RANDOM-STATE-P [*object*]	Numerical	Function	150
RASSOC [*key*, *a-list*, *test*]	A-list	Function	96
RASSOC-IF [*test*, *a-list*]	A-list	Function	96
RATIONAL	IRRATNAL.LSP	Keyword	286
RATIONALP [*object*]	Recognizer	Function	41
READ [*source*, *eof-p*, *eof-v*]	Input	Function	207
READ-ATOM [*source*, *eof-p*, *eof-v*]	Input	Function	208
READ-BASE: 10	Input	Control var.	218
READ-BYTE [*source*, *eof-p*, *eof-v*]	Input	Function	209
READ-CHAR [*source*, *eof-p*, *eof-v*]	Input	Function	209
READ-DOT [*x*, *y*]	Screen	Function	237
READ-FROM-STRING [*string*, *eof-p*, *eof-v*]	Input	Function	212
READ-LINE [*source*, *eof-p*, *eof-v*]	Input	Function	209
READ-RECORD [*n*, *source*, *eof-p*, *eof-v*]	Input	Function	209
READ-STRINGS	Input	Variable	212
READ-UPCASE: T	Input	Control var.	218
RECLAIM [ ]	Memory	Function	200
REDUCE [*function*, *list*, *initial*]	Mapping	Function	196

REGISTER [*regnum*, *low*, *high*]	Hardware	Function	246
REM [*n*, *m*]	Numerical	Function	142
REMD [*symbol*]	Definition	Function	117
REMF [*place*, *key*]	Property	Function	110
REMFLAG [*symbol*, *flag*]	Flag	Function	111
REMOVE [*item*, *list*, *test*]	Deletion	Function	77
REMOVE-DUPLICATES [*list*, *test*]	Deletion	Function	79
REMOVE-IF [*test*, *list*]	Deletion	Function	78
REMPROP [*symbol*, *key*]	Property	Function	110
RENAME-FILE [*oldname*, *newname*]	File	Function	262
REPLACE [$list_1$, $list_2$]	Modifier	Function	71
REPLACE-NTH [*item*, *list*, *n*]	Modifier	Function	75
:REQUIRED-FLAVORS [$flavor_1$, ..., $flavor_n$]	FLAVORS.LSP	Option	298
REST [*list*]	Selector	Function	56
RESTART [ ]	Control	Function	182
RETURN [*form*]	Control	Special form	181
REVAPPEND [*list*, *object*]	Constructor	Function	68
REVERSE [*list*, *object*]	Constructor	Function	68
RIGHT [*angle*]	GRAPHICS.LSP	Function	310
RIGHT-BUTTON	MOUSE.LSP	Variable	315
ROUND [*n*, *m*]	Numerical	Function	141
ROW [ ]	Screen	Function	231
ROWS [ ]	Screen	Function	231
RPLACA [obj_1, obj_2]	Modifier	Function	70
RPLACD [obj_1, obj_2]	Modifier	Function	70
RUN-WINDOW	WINDOWS.LSP	Keyword	340
SAVE [*file*]	Memory image	Function	266
SEC [*n*]	IRRATNAL.LSP	Function	290
SECH [*n*]	IRRATNAL.LSP	Function	292
SECOND [*list*]	Selector	Function	56
SELF	FLAVORS.LSP	Variable	303
SEND [*object*, *operation*, arg_1, ..., arg_n]	FLAVORS.LSP	Macro	303
SET [*symbol*, *object*]	Symbol	Function	102
SET-CURSOR [*row*, *column*]	Screen	Function	231
SET-DIFFERENCE [$list_1$, $list_2$, *test*]	Set	Function	91
SET-DISPLAY [ ]	WINDOWS.LSP	Function	342
SET-EXCLUSIVE-OR [$list_1$, $list_2$, *test*]	Set	Function	92
SET-IN-CLOSURE [*closure*, *symbol*, *object*]	CLOSURE.LSP	Function	307

SET-INSERT [ ]	WINDOWS.LSP	Function 342
SET-MACRO-CHAR [*char*, *defn*, *flag*]	Input	Function 215
SET-MENU-COLOR [ ]	WINDOWS.LSP	Function 342
SET-MUTE [ ]	WINDOWS.LSP	Function 342
SET-WORK-COLOR [ ]	WINDOWS.LSP	Function 342
SETCOLOR [*angle*]	GRAPHICS.LSP	Function 310
SETF [$place_1$, $form_1$, ..., $place_n$, $form_n$]	Symbol	Macro 105
SETHEADING [*angle*]	GRAPHICS.LSP	Function 310
SETPOS [*x*, *y*]	GRAPHICS.LSP	Function 310
SETQ [$symbol_1$, $form_1$, ..., $symbol_n$, $form_n$]	Symbol	Special form 103
:SETTABLE-INSTANCE-VARIABLES	FLAVORS.LSP	Option 298
SEVENTH [*list*]	Selector	Function 56
SHIFT [*n*, *count*]	Numerical	Function 146
SHOW-LOAD: T	DEBUG.LSP	Control var. 335
SHOW-MOUSE [ ]	MOUSE.LSP	Function 314
SHOW-PROMPT [*prompt*]	WINDOWS.LSP	Function 342
SIGNUM [*n*]	Numerical	Function 138
SIN [*n*]	IRRATNAL.LSP	Function 290
SIN-DEG [*angle*]	GRAPHICS.LSP	Function 309
SINH [*n*]	IRRATNAL.LSP	Function 292
SIXTH [*list*]	Selector	Function 56
SLEEP [*seconds*]	Hardware	Function 240
SNAPSHOT [*address*, *atom*]	Hardware	Function 248
SOME [*test*, $list_1$, ..., $list_n$]	Mapping	Function 194
SORT [*list*, *test*]	Sorting	Function 100
SPACES [*n*, *sink*]	Output	Function 223
SPECIAL	Definition	Keyword 112
SPECIAL-FORM-P [*symbol*]	Definition	Function 114
SPLIT [*list*]	Sorting	Function 98
SQRT [*n*]	IRRATNAL.LSP	Function 287
STABLE-SORT [*list*, *test*]	Sorting	Function 100
STACK-LIST [ ]	Constructor	Function 69
STRING-CAPITALIZE [*atom*]	String	Function 165
STRING-DOWNCASE [*atom*]	String	Function 165
STRING-EQUAL [$atom_1$, $atom_2$]	String	Macro 161
STRING-GREATERP [$atom_1$, $atom_2$]	String	Macro 163
STRING-INDEX	Input	Variable 212
STRING-LEFT-TRIM [*chars*, *atom*]	String	Function 164
STRING-LESSP [$atom_1$, $atom_2$]	String	Macro 163

STRING-NOT-EQUAL [$atom_1$, $atom_2$]	String	Macro	163
STRING-NOT-GREATERP [$atom_1$, $atom_2$]	String	Macro	163
STRING-NOT-LESSP [$atom_1$, $atom_2$]	String	Macro	163
STRING-RIGHT-TRIM [*chars*, *atom*]	String	Function	164
STRING-TRIM [*chars*, *atom*]	String	Function	164
STRING-UPCASE [*atom*]	String	Function	165
STRING= [$atom_1$, $atom_2$, *flag*]	String	Function	161
STRING< [$atom_1$, $atom_2$, *flag*]	String	Function	162
STRING> [$atom_1$, $atom_2$, *flag*]	String	Function	162
STRING<= [$atom_1$, $atom_2$, *flag*]	String	Function	162
STRING>= [$atom_1$, $atom_2$, *flag*]	String	Function	162
STRING/= [$atom_1$, $atom_2$, *flag*]	String	Function	162
STRINGP [*object*]	Recognizer	Function	40
SUB1 [*n*]	Numerical	Function	133
SUBLIS [*a-list*, *object*, *test*]	A-list	Function	97
SUBLIST [*list*, *n*, *m*]	Constructor	Function	65
SUBSETP [$list_1$, $list_2$, *test*]	Set	Function	93
SUBST [*new*, *old*, *object*, *test*]	Substitution	Function	87
SUBST-IF [*new*, *test*, *object*]	Substitution	Function	88
SUBSTITUTE [*new*, *old*, *list*, *test*]	Substitution	Function	85
SUBSTITUTE-IF [*new*, *test*, *list*]	Substitution	Function	86
SUBSTRING [*atom*, *n*, *m*]	String	Function	160
SYMBOL-FUNCTION [*symbol*]	Definition	Function	113
SYMBOL-PLIST [*symbol*]	Property	Function	109
SYMBOL-VALUE [*symbol*]	Symbol	Function	101
SYMBOLP [*object*]	Recognizer	Function	40
SYMEVAL-IN-CLOSURE [*closures*, *symbol*]	CLOSURE.LSP	Function	307
SYSTEM [*n*]	Control	Function	182
TAB-EXPAND: NIL	Input	Control var.	218
TAILP [$list_1$, $list_2$]	Comparator	Function	48
TAN [*n*]	IRRATNAL.LSP	Function	290
TANH [*n*]	IRRATNAL.LSP	Function	292
TCONC [*dotted-pair*, *object*]	Modifier	Function	76
TENTH [*list*]	Selector	Function	56
TERPRI [*n*, *sink*]	Output	Function	223
THIRD [*list*]	Selector	Function	56
THROW [*tag*, *form*]	Control	Special form	181
THROW-TAG	Control	Variable	181

TIME [*reset*]	Hardware	Function	238
TIME-ZONE: 0	Hardware	Control var.	239
TONE [*frequency, milliseconds*]	Hardware	Function	240
TREE-EQUAL [obj_1, obj_2, *test*]	Comparator	Function	47
TRUNCATE [*n, m*]	Numerical	Function	141
TURTLE [ ]	GRAPHICS.LSP	Function	312
TURTLE-WINDOW [ ]	GRAPHICS.LSP	Function	312
:TYPE LIST	STRUCTUR.LSP	Option	296
UNDEFINED-HOOK: 'UNDEFINED-FUNCTION	Break	Control var.	276
UNDEFINED-FUNCTION [*form*]	Break	Function	276
UNDEFMETHOD [(*flavor type method*)]	FLAVORS.LSP	Macro	300
UNDERFLOW [*n*]	Numerical	Function	124
UNION [$list_1$, $list_2$, *test*]	Set	Function	89
UNLESS [*testform*, $form_1$, ..., $form_n$]	Control	Macro	173
UNPACK [*atom*]	String	Function	158
UNREAD-CHAR [*source*]	Input	Function	210
UNWIND-PROTECT [$form_1$, $form_2$, ..., $form_n$]	Control	Special form	180
UPDATE-STATE [*arglist*]	WINDOWS.LSP	Function	343
UPDATE-WINDOW	WINDOWS.LSP	Keyword	340
UPPER-CASE-P [*symbol*]	Character	Function	151
VECTOR [obj_1, ..., obj_n]	ARRAY.LSP	Function	282
VIDEO-MODE [*n*]	Screen	Function	233
WINDOW-TYPES	WINDOWS.LSP	Control var.	339
WINDOWS [*command-line*]	WINDOWS.LSP	Function	343
WHEN [*testform*, $form_1$, ..., $form_n$]	Control	Macro	173
WRITE [*object, sink*]	Output	Function	220
WRITE-BYTE [*n, m, sink*]	Output	Function	224
WRITE-LINE [*symbol, sink*]	Output	Function	222
WRITE-STRING [*symbol, sink*]	Output	Function	222
Y-OR-N-P [*message*]	Query	Function	278
YES-OR-NO-P [*message*]	Query	Function	278
ZAP-STRING [*symbol*]	Memory	Function	203
ZEROP [*object*]	Numerical	Function	125

+ $[n_1, ..., n_m]$	Numerical	Function	131
- $[n_1, ..., n_m]$	Numerical	Function	131
* $[n_1, ..., n_m]$	Numerical	Function	132
/ $[n_1, ..., n_m]$	Numerical	Function	132
+TRAP $[obj_1, obj_2]$	Break	Function	277
-TRAP $[obj_1, obj_2]$	Break	Function	277
*TRAP $[obj_1, obj_2]$	Break	Function	277
/TRAP $[obj_1, obj_2]$	Break	Function	277
= $[n_1, ..., n_m]$	Numerical	Function	127
/= $[n_1, ..., n_m]$	Numerical	Function	127
< $[n_1, ..., n_m]$	Numerical	Function	128
> $[n_1, ..., n_m]$	Numerical	Function	128
<= $[n_1, ..., n_m]$	Numerical	Function	129
>= $[n_1, ..., n_m]$	Numerical	Function	129
\ (back slash)	Input	Escape char.	207
| (vertical bar)	Input	Escape char.	207
((left parenthesis)	Input	Macro char.	216
) (right parenthesis)	Input	Macro char.	216
] (right square bracket)	Input	Macro char.	216
, (comma)	Input	Macro char.	216
' (single quote)	Input	Macro char.	216
; (semicolon)	Input	Macro char.	216
" (double quote)	Input	Macro char.	217
` (backquote)	Input	Macro char.	217
+	Driver	Variable	271
++	Driver	Variable	271
+++	Driver	Variable	271
*	Driver	Variable	271
**	Driver	Variable	271
***	Driver	Variable	271
-	Driver	Variable	271
/	Driver	Variable	271

Concept Index

A-list 94, 308, 332
Abort option 33, 273
ABS 138
Absolute value 138
ABSTRAP 277
ACONS 94
ACOS 291
ACOSH 293
ACOT 291
ACOTH 293
ACSC 291
ACSCH 293
Active display page 235
Active selection field 321
Actual arguments 189
Adapter cards 371
ADD1 133
Address space 31
Address typed data structures 2
Address typing 360
ADDRESS.LSP 6, 339
ADJOIN 61
ALLOCATE 204
Alpha option 322, 326
ALPHA-CHAR-P 152
Alphabetic characters 152
ALPHANUMERICP 152
Alt key 11
Alt-! 331
Alt-C 331
Alt-D 331
Alt-R 331
Alt-S 331
Alt-T 331
AND 51
Animal game 5, 23
ANIMAL.LSP 7, 23
ANIMAL.MEM 7, 23
APPEND 63
APPLY 190
Arccosecant 291
Arccosine 291
Arccotangent 291
Arcsecant 291
Arcsine 291
Arctangent 291
AREF 283
Argument passing 362
Arithmetic operators 2, 131
Array options 282
ARRAY-DIMENSION 284
ARRAY-DIMENSIONS 284
ARRAY-IN-BOUNDS-P 285
ARRAY-RANK 284
ARRAY-TOTAL-SIZE 284
ARRAY.LSP 6
ARRAYP 285
Arrays 4, 281
Arrow keys 326
Artificial intelligence (AI) iii, 367
ASCII 156
ASCII characters 28, 156, 224
ASEC 291
ASECH 293
ASH 146
ASIN 291
ASINH 293
Assembly language listing 390
ASSOC 95, 308
ASSOC-IF 95
Association list 94, 308, 332
AT&T 6300 computer 371
ATAN 291
ATANH 293
Atom 28, 42
Atom space 31, 360
Attributes, character 373
Author option 335, 337, 338
Auto-quoting 17, 28
Auxiliary functions 39

BACK 311

BACKGROUND-COLOR 234
Backquote macro 217
Backslash character 207, 249
Backspace key 11, 15, 321, 329
Backtrace option 335, 338
Base column 230
Base page 244, 359, 382
Base row 230
BENCHMARK 238
Big integers 29
Binary trees 30
BINARY-LOAD 247
Bit-wise logical operators 144
Blinking 373
Blinking parentheses 319
Block Begin option 322
Block Copy option 322
Block Delete option 322
Block End option 322
Block Jump Begin option 322
Block Jump End option 322
Block Move option 322
Block Write option 322
Boolean operations 50
Border color 372, 373
BORDER-COLOR 234
BOUNDP 101
BOUNDP-IN-CLOSURE 308
BREAK 273
Break characters 208
Break option 33, 273
Break option menu 33
BREAK-CHARS 208
BUTLAST 66
BUTTON-PRESSES 315
BUTTON-RELEASES 315

CAAAAR 54
CAAADR 54
CAAAR 54
CAADAR 54
CAADDR 54
CAADR 54
CAAR 54
CADAAR 54
CADADR 54
CADAR 54
CADDAR 54
CADDDR 54
CADDR 54
CADR 54
Calender time 239
Calling convention 363
Calling machine-coded routines 363
CAR 53
Car branch 53
Car cell 27, 53
CASE 175
Case conversion 226
Case insensitive 155, 161
Case sensitive 154, 161, 324, 330
CATCH 179
CDAAAR 54
CDAADR 54
CDAAR 54
CDADAR 54
CDADDR 54
CDADR 54
CDAR 54
CDDAAR 54
CDDADR 54
CDDAR 54
CDDDAR 54
CDDDDR 54
CDDDR 54
CDDR 54
CDR 53
Cdr branch 53
Cdr cell 27, 53
CEILING 140
CGA graphics adapter 8, 371
Chain-loading 19
CHANGE-WINDOW 341
CHAR 160
CHAR-CODE 156
CHAR-DOWNCASE 157
CHAR-EQUAL 155
CHAR-GREATERP 155

CHAR-LESSP 155
CHAR-NOT-EQUAL 155
CHAR-NOT-GREATERP 155
CHAR-NOT-LESSP 155
CHAR-UPCASE 157
CHAR/= 154
CHAR< 154
CHAR<= 154
CHAR= 154
CHAR> 154
CHAR>= 154
Character 151
Character and string processing 2
Character attributes 373
Character functions 151
Character generator table 387
Chronological ordering 49
Circular list 72, 193
CLEAR-INPUT 211
CLEAR-SCREEN 232
CLOSE-INPUT-FILE 251
CLOSE-OUTPUT-FILE 257
CLOSE-WINDOW 340
Closed interval 129
Closed pointer universe 2
CLOSURE 304
CLOSURE-ALIST 308
CLOSURE-FUNCTION 308
CLOSURE-VARIABLES 308
CLOSURE.LSP 6
CLOSUREP 306
Closures 4, 304
CLR 310
Code segment 361
Code space 31
CODE-CHAR 156
Color, background 234
Color, border 234
Color, foreground 234
Color, palette 234
Color, plot 237
Color graphics adapter (CGA) 8, 248
Color graphics display monitor 236
Color monitor 8
Color numbers 237, 372
COLUMN 231
COLUMNS 231
COM file 2, 4, 6, 266
COMn device names 220, 256
Command line 19, 20, 35, 269
Command mode 321
Command processor 183, 265
COMMENT 188, 215
Comment character 14
Comment macro characters 214
Common LISP 4
COMMON.LSP 6
Communication facilities 2, 206
Comparators 45
Compatible compilers 356
Compilation procedure 349
COMPILE 346
COMPILE.LSP 7, 345
COMPILEn.LSP 7, 345
Compile time 199
COMPILE-FILE 348
Compiled function behavior 356
Compiler 4, 7, 345
Compiler errors 354
Component extraction 138
COMPRESS 159
Computer memory 8
Computer type 229, 386
COMTOEXE.COM 6, 266
COND 17, 174
Condensing 120
Conditional branching 169
Conditionals 173
CONS 60
Cons space 31, 203, 360
Conses 30, 31, 43
Console bell character 385
Console interrupt 33, 34, 276
Console screen 229
CONSOLE-INTERRUPT 276
CONSP 43
Constant form 187
CONSTANTP 187

Constructors 60
Continue option 33, 273
CONTINUE-PROMPT 341
Control characters 156
Control constructs 3, 169
Control keys 15, 326, 387
Control stack 31
Control variable 39
COPY 365, 390
COPY-ALIST 94
COPY-CHAR-TYPE 214
COPY-CLOSURE 306
COPY-LIST 64
COPY-TREE 64
Copyright notice ii, 9
COS 290
COS-DEG 309
Cosecant 290
COSH 292
Cosine 290
COT 290
Cotangent 290
COTH 292
COUNT 82
COUNT-IF 82
CREATE-WINDOW 340
CSC 290
CSCH 292
CSMEMORY 243
Ctrl key 11
Ctrl-A 15, 327
Ctrl-C 328
Ctrl-D 15, 327
Ctrl-E 327
Ctrl-End 327
Ctrl-Enter 327
Ctrl-F 15, 327
Ctrl-G 15, 329
Ctrl-H 15, 329
Ctrl-Home 327
Ctrl-I 16, 327, 329
Ctrl-J 327
Ctrl-K 0 328
Ctrl-K 1 328
Ctrl-K 2 328
Ctrl-K 3 328
Ctrl-K A 331
Ctrl-K B 330
Ctrl-K C 330
Ctrl-K D 331
Ctrl-K K 330
Ctrl-K Q 331
Ctrl-K R 331
Ctrl-K S 331
Ctrl-K V 330
Ctrl-K W 330
Ctrl-K Y 330
Ctrl-L 330
Ctrl-Left arrow 15, 327
Ctrl-M 15, 327, 329
Ctrl-N 329
Ctrl-O B 330
Ctrl-O C 330
Ctrl-O D 330
Ctrl-O E 330
Ctrl-O J B 330
Ctrl-O J E 330
Ctrl-O M 330
Ctrl-O W 330
Ctrl-P 16, 329
Ctrl-PgDn 328
Ctrl-PgUp 328
Ctrl-Q 327
Ctrl-Q 0 328
Ctrl-Q 1 328
Ctrl-Q 2 328
Ctrl-Q 3 328
Ctrl-Q A 330
Ctrl-Q B 330
Ctrl-Q Backspace 16, 329
Ctrl-Q C 328
Ctrl-Q D 15, 327
Ctrl-Q E 327
Ctrl-Q F 330
Ctrl-Q H 16, 329
Ctrl-Q I 327
Ctrl-Q K 330
Ctrl-Q L 330

Ctrl-Q P 327
Ctrl-Q R 328
Ctrl-Q Rub 16, 329
Ctrl-Q S 15, 327
Ctrl-Q T 329
Ctrl-Q V 331
Ctrl-Q X 327
Ctrl-Q Y 16, 329
Ctrl-R 328
Ctrl-Right arrow 15, 327
Ctrl-S 15, 327
Ctrl-T 15, 329
Ctrl-U 16, 329
Ctrl-V 16, 329
Ctrl-W 328
Ctrl-X 327
Ctrl-Y 16, 329
Ctrl-Z 328
Ctrl-\ 15
Ctrl-] 15
Ctrl-^ 331
Ctrl-^ ! 331
Ctrl-^ C 331
Ctrl-^ D 331
Ctrl-^ R 331
Ctrl-^ S 331
Ctrl-^ T 331
Current directory 249
Current window 230
Cursor 15, 236, 326
CURSOR-LINES 236
Cycles per second 240

Data cells 27
Data compaction 32
Data entry fields 321
Data objects 27, 40
Data representation 360
Data space reallocation 32
Data spaces 31, 388
Data structures 27
Data types 294
DEBUG.LSP 6, 318
Debugger 4, 335
DECF 135
Decimal notation 121, 228
Decoded time 239
DECQ 134
Decrement 134, 135
Default drive 249
Default line length 384
Default Read Table 34, 207, 382
DEFAULT-DRIVE 264
DEFAULT-PATH 264
DEFFLAVOR 298
Defflavor options 298
Definition primitives 28, 112
DEFMACRO 39, 119
DEFMETHOD 300
DEFSTRUCT 294
Defstruct options 294
DEFUN 39, 118
DEFWHOPPER 301
Del key 11, 15, 329
DELETE 77
Delete Block option 322
Delete Character option 322
Delete Line option 322
Delete option 335
Delete Right-end option 322
Delete Word option 322
DELETE-COMPILER 347
DELETE-DUPLICATES 79
DELETE-FILE 262
DELETE-IF 78
DELETE-LINES 232
DELETE-NTH 75
Deletion functions 77
Demonstration game programs 5, 7
DENOMINATOR 139
DENTRAP 277
Destructuring 119
Device names 220, 249
DIGIT-CHAR-P 153
Dimension, array 284
Dimension Error 34, 282
DIRECTORY 264
Discrimination net 23

Disk Full 34, 220
Disk operating system 8
Diskette drive 8, 10
DISPLAY-PAGE 235
DIVIDE 143
DO 177
DO* 177
Doctor game 5, 23
DOCTOR.LSP 7, 23
Documentation 5
DOLIST 178
Dollar sign 271
DOS 183
DOT 309
Dot notation 30, 220
DOTIMES 178
Dotted-pairs 30
Double precision 122
Down arrow key 11, 327
DRIVER 270, 271, 343
Driver function 269
DSMEMORY 243
Dynamic binding 297
Dynamic closure 304

EDIT.LSP 6, 318
Editor 3, 319
Editor control variables 333
Editor screen 320
EGA graphics adapter 8, 371
EIGHTH 56
Eights puzzle 5, 23
EIGHTS.LSP 7, 23
Eliza 23
EMACS-like editor 318, 332, 380
Empty list 30, 44
ENCODE-UNIVERSAL-TIME 239
End key 11, 15, 327
End-Of-File 34, 206
End-of-file error processing 206
ENDP 44
Enhanced color monitor 8
Enhanced graphics adapter (EGA) 8
Enter key 11, 14, 15, 321, 327, 329
Entry mode 16
Environment parameter 265
EOF character 385
EQ 45
EQL 46
EQUAL 47
Error beep 322, 323, 336
Error break 121
Error messages 34, 354
ERROR-BEEP 341
Esc key 11, 33, 34, 321, 326
Escape characters 156, 207, 226
EVAL 184
Eval function 184
EVAL-FUNCTION-P 114
Eval-quote LISP 270
EVENP 126
EVERY 195
Exact mode 35, 286
EXACT-MODEP 286
EXE file 4, 6, 266
Executable files 6
EXECUTE 183
EXECUTE-OPTION 341
Exit code 182, 385
EXP 286
EXPLODE 158
EXPT 287
Extended COND 17
Extended Function Key Table 279, 388
Extended function keys 279, 376, 388

F3 function key 16
Factorial 17
False 40, 50
FBOUNDP 114
FIFTH 56
File and directory 262
File compiler 345
File not found 19, 21, 35
File system 2, 249
FILE-DATE 263
FILE-LENGTH 263
FILE-LINE-LENGTH 261

FILE-READ-POSITION 254
FILE-WRITE-POSITION 260
FIND 80
FIND-IF 80
FIRST 56
FIRSTN 66
FLAG 111
FLAGP 111
Flags 28, 111
Flavors 4, 297
FLAVORS.LSP 6
Floating slash 121
FLOOR 140
FMAKUNBOUND 117
FOREGROUND-COLOR 234
Form evaluation 184
Formal arguments 189
Formal definition 39
Formfeed character 207, 208
FORWARD 311
FOURTH 56
FRESH-LINE 223
Full condensing 120
FUNCALL 189
Function application 189
Function body 189
Function cell 28
Function definition 28
Function keys 376
Function mapping 2
FUNCTIONP 187
Functions 362

Garbage collection 32, 69, 200, 388
Garbage collection statistics 201
GC cursor indication 386
GCD 136
GCDTRAP 277
Generic MS-DOS 3
GENSYM 168
GET 109
GET-DECODED-TIME 239
GET-MACRO-CHAR 215
GET-PROPERTIES 109
GET-UNIVERSAL-TIME 239
GETD 112
GETF 109
GETSET 265
Global storage area 244, 359, 389
GMT 239
GO 17
GO-DOS 342
Graphics 4
Graphics character 386
GRAPHICS.LSP 6
Greatest common divisor 136
Greenwich Mean Time 239

HANOI.LSP 7, 24
Hard disk drive 9
Hardware interface 238
Hash link 29
Hashing algorithm 2, 28
Header message 229, 386
Help facility 5, 24
HELP.LSP 6, 24
Hercules compatible graphics 233
Hercules graphics adapter 8, 371
HERCULES.COM 6, 233
Hertz 240
HIDE-MOUSE 314
Highlight 321
History option 335, 338
Holes 34
Home key 11, 15, 327
Home position 230
HP 95LX computer 371
Hyperbolic arccosecant 293
Hyperbolic arccosine 293
Hyperbolic arccotangent 293
Hyperbolic arcsecant 293
Hyperbolic arcsine 293
Hyperbolic arctangent 293
Hyperbolic cosecant 292
Hyperbolic cosine 292
Hyperbolic cotangent 292
Hyperbolic secant 292
Hyperbolic sine 292

Hyperbolic tangent 292

IBM Personal Computer 8
IDENTITY 186
Identity cell 29
IF 173
Implicit COND 17, 169, 170, 174
Implicit PROGN 17, 169
IMPLODE 159
In-line machine code 352
INCF 135
Include Error 35, 296
Incompatible Definition 35, 294
INCQ 134
Increment 134, 135
Incremental compiler 345
Indicator key 94
Initable flavors 298
Input control variables 218
Input files 250
Input functions 206
Input radix base 218
INPUT-FILE 252
INPUT-FILE-P 252
INPUT-FILES 252
Ins key 11, 16, 329
Insert mode 16, 323, 328
INSERT-LINES 232
INSERT-NTH 75
Instance variables 297
Insufficient Arguments 35, 121
Insufficient Memory, Aborting 21, 35
INTEGER-LENGTH 147
INTEGERP 41
Integers 29
Intel 8086 8, 31, 359
Intensity 373
Internal Errors 355
Interpreter 184
Interpreter overhead 351
INTERRUPT 246
Interrupt and error handling 273
Interrupt sensitivity 385
INTERSECTION 90

Invalid Argument 35
Invalid Place Form 35, 135
Inverse video 321
Irrational functions 4, 286
IRRATNAL.LSP 6
ISQRT 287
Iteration 176

Jump Bottom option 322
Jump End option 322
Jump Last option 322
Jump Start option 322
Jump Top option 323

KANJI characters 386
Keyword Not Handled 35, 302
Kilobytes 2

LAMBDA 29, 112
LAMBDA bodies 17
LAST 57
LCM 137
LCMTRAP 277
LCONC 76
LDIFF 67
Least common multiple 137
LEFT 310
Left arrow key 11, 15, 327
LENGTH 59
Length cell 29
LESSONS.LSP 7
LET 17, 172
LET* 17, 172
LET-CLOSED 305
Lexical binding 297
Lexicographical comparison 154, 162
LINE 309
Line edit mode 206
Line editing 15
Line editor 206, 279
Line Editor Control Key Table 279, 387
LINE-EDIT 279
Linefeed character 207, 208
LISP books 369

LISP periodicals 368
LISP professional societies 367
LISP programming language iii
LIST 62
List Back option 323
List Delete option 323
List Eval option 323
List Forward option 323
List merge sort 100
List Next option 323
List notation 30, 220
List processing 2
LIST* 62
LIST-LENGTH 59
LISTEN 211
LISTP 43
Lists 30, 43, 53
LN 288
LOAD 19, 20, 268
Load option 335
Loading and linking 364
Local variables 17, 101
LOCATE-IN-CLOSURE 307
LOCATION 245
LOG 288
Logical and 145
Logical end of a file 385
Logical equivalence 145
Logical exclusive or 145
Logical inclusive or 145
Logical not 144
Logical operators 50
LOGO computer language 309
LOOP 17, 176
LOWER-CASE-P 151
Lowercase characters 151
LPTn device names 220, 256

Machine code 28
Machine language 112
Machine language kernel 2
Machine language routines 3
MACRO 29, 112
Macro call 197
Macro Character Function Table 383
Macro characters 2, 207
Macro expansion 184, 197
MACRO-FUNCTION 197
MACRO-P 114
MACROEXPAND 198, 199
MACROEXPAND-1 198
Macros 3, 119, 184
MAKE-ARRAY 282
MAKE-INSTANCE 302
MAKE-LIST 64
MAKE-WINDOW 230
MAPC 191
MAPCAN 193
MAPCAR 192
MAPCON 193
MAPL 191
MAPLIST 192
Mapping functions 191
Mapping predicates 194
MAX 130
Maximum paragraph address 384
MAXTRAP 277
MCGA graphics adapter 371
MDA graphics adapter 8, 371
MEMBER 84
MEMBER-IF 84
MEMORY 242
Memory Full 31, 35, 201
Memory image files 3, 20, 266
Memory management 3, 31, 200
Memory reallocation 200
Menu options 320, 321
Merging and sorting 2
Metamind game 5
METAMIND.LSP 7, 24
Method combination 297
Methods 297
Milliseconds 240
MIN 130
MINTRAP 277
MINUSP 125
MISMATCH 83
Missing Required Flavor 36, 302

MOD 142
Mode option 323, 335, 338, 342
Modifier functions 28
Modifiers 70
Modulus 142
Monitor types 371
Monochrome display adapter 8, 248
Monochrome monitor 8
Mouse cursor display byte 314
Mouse interface 4, 313
MOUSE-BOX 316
MOUSE-POSITION 314
MOUSE-RESET 313
MOUSE-STATUS 315
MOUSE.LSP 6
MOVD 116
MS-DOS 8
muLISP diskette files 6
muLISP Reference Manual 5
MULISP subdirectory 9, 13
MULISP.COM 2, 6, 265
MULISPn.LES 7, 25
Multi-sync display monitor 236
Multiple escape characters 207, 385
Multisync color monitor 8

Name conflicts 357
Named record structure 294
NBUTLAST 74
NEC PC-9801 Personal Computer 8
NECGRAPH.LSP 6
NEQ 45
NEQL 46
NEW-CODE-SPACE 205
Newline character pairs 223
Newton's method 287
NIL 28, 29, 30, 40, 44, 184
NINTERSECTION 90
NINTH 56
NLAMBDA 29, 112
No-eval function 184
NO-EVAL-FUNCTION-P 114
No-spread function 189
Non-list object 44
Non-local exits 179
Non-terminating macro characters 214
Noninteger Argument 36, 121
NonNIL 40, 50
Nonnumeric Argument 36, 121
Nonresident editor 317
Nonsymbolic Argument 36, 294
NOT 50
NOTANY 194
NOTEVERY 195
NSET-DIFFERENCE 91
NSET-EXCLUSIVE-OR 92
NSUBLIS 97
NSUBST 87
NSUBST-IF 88
NSUBSTITUTE 85
NSUBSTITUTE-IF 86
NTH 58
NTHCDR 58
NULL 44
Null list 30, 44
Number vectors 29, 31, 201
NUMBERP 41
Numbers 29, 121
NUMERATOR 139
Numeric characters 152
Numeric weights 153
NUMERIC-CHAR-P 152
Numerical predicates 125
NUMTRAP 277
NUNION 89

Object data 27, 40
Object, flavor 297
Object list 69
OBLIST 69
ODDP 126
Open interval 128
OPEN-INPUT-FILE 250
OPEN-OUTPUT-FILE 255
Option lines 320
Options Color Menu option 323, 336, 342
Options Color Work option 323, 336, 342
Options Display option 321, 323, 336, 342

Options Execute option 323, 336, 342
Options Mute option 322, 323, 336, 342
Options submenu 342
OR 52
ORDERP 49
OTHERWISE 175
Output control variables 225
Output device 220, 255, 324
Output files 255
Output functions 220
Output radix base 227
OUTPUT-FILE 258
OUTPUT-FILE-P 258
OUTPUT-FILES 258
Overwrite mode 16, 323, 328

PACK 159
PACK* 159
PAIRLIS 94
PALETTE-COLOR 234
Parity bit mask 385
PARSE-INTEGER 213
Pathname 19, 20, 249
PC compatible computer 8
PC-9801 compatible computer 8
PDS.SYS 318
PEEK-CHAR 210
PENDOWN 310
PENUP 310
Perfect compilers 356
Performance considerations 351
Personal System/2 Computer 8
PgDn key 11, 328
PgUp key 11, 328
Physical end of a file 385
PI 289
Pixels 230, 237
Place forms 105, 135
PLOT-CIRCLE 237, 386
PLOT-DOT 237, 386
PLOT-LINE 237, 386
PLUSP 125
Pname cell 28
Point 332
POP 106
PORTIO 241
POSITION 81
POSITION-IF 81
Potential problems 356
PPRINT 221
PRECISION 122
Predicates 40
Preface iii
PRIN1 30, 221, 385
PRINC 30, 221
PRINT 30, 221
Print Layout option 323, 334, 336
Print name string 28
Print names 31, 151, 203
Print Options option 324, 336
Print Printer option 323
Print Printer option 336
PRINT-LENGTH 167
Printed record 226
Printer 220, 225, 256, 334
Printer control variables 334
PRN device name 256
PROBE-FILE 262
Problem Report Form ii, 9
PROG 17
PROG1 171
PROG2 171
PROGN 17, 170
Program Development System 6, 21, 317
Program error 33
Programming 17
Prompt 14, 271
Prompt line 320
Prompt string 386
PROMPT-YN 341
Property list 28, 108
Property list cell 28
PSETQ 104
Pseudo-code 28, 31
Pseudo-code compiler 3
Pseudo-registers 383
PUSH 107
PUSHNEW 107

PUT 108
PUTD 115

Quit option 324, 336, 342
QUIT-PROGRAM 342
QUOTE 186

Radix base 158, 218, 227
Random numbers 148
Rank, array 284
RASSOC 96, 308
RASSOC-IF 96
RATIONAL 286
Rational arithmetic 121
Rational notation 121, 228
RATIONALP 41
Ratios 29
Raw input mode 206
READ 30, 207
Read macro characters 214
Read preserving white space 385
READ-ATOM 208
READ-BYTE 209, 376
READ-CHAR 209
READ-DOT 237
Read-eval-draw loop 312
Read-eval-print loop 14, 226, 271
READ-FROM-STRING 212
READ-LINE 209
Read-only memory 229
READ-RECORD 209
README 6
Real time clock 238
Real-time applications 32
Reallocation sensitivity 384
RECLAIM 200
Recognizers 27, 40
Record structures 294
Recursive definition 17
Recursive implicit COND 170
REDUCE 196
REGISTER 246
REM 142
REMF 110
Remainder 142
REMD 117
REMFLAG 111
REMOVE 77
REMOVE-DUPLICATES 79
REMOVE-IF 78
REMPROP 110
RENAME-FILE 262
Replace option 324
REPLACE-NTH 75
Resident editor 317
REST 56
RESTART 182
Restart option 33, 274
Resume option 336, 338
RETURN 181
Return character 207, 208
Return code 182
REVAPPEND 68
REVERSE 68
RIGHT 310
Right arrow key 11, 15, 327
ROM 229
ROUND 141
ROW 231
ROWS 231
RPLACA 70
RPLACD 70
Rub key 15, 329
Run time 199
RUN-WINDOW 340
Runtime System, muLISP 4, 266

SAVE 20, 266
Screen colors 372
Search functions 80
Search option 324
SEC 290
Secant 290
SECH 292
SECOND 56
Segments 360
Selection fields 321
Selectors 53

SELF 303
Semicolons 14
SEND 303
Sequencing 169
Serial number 9, 13
SET 102
Set functions 89
SET-CURSOR 231
SET-DIFFERENCE 91
SET-DISPLAY 342
SET-EXCLUSIVE-OR 92
SET-IN-CLOSURE 307
SET-INSERT 342
SET-MACRO-CHAR 215
SET-MENU-COLOR 342
SET-MUTE 342
SET-WORK-COLOR 342
SETCOLOR 310
SETF 105
SETHEADING 310
SETPOS 310
SETQ 103
Sets 89
SEVENTH 56
Shallow variable binding 2
SHIFT 146
Shift key 11
Shift-Tab 321, 327
SHOW-MOUSE 314
SHOW-PROMPT 342
Side-effects 70, 101
Sign cell 29
SIGNUM 138
SIN 290
SIN-DEG 309
Sine 290
Single escape characters 19, 20, 207, 385
Single precision 122
SINH 292
SIXTH 56
SLEEP 240
Slot description 294
Small integers 29
SNAPSHOT 248
Soft Warehouse, Inc. iv
SOME 194
SORT 100
Sorting functions 98
Source file library 4, 6, 281
Source files 19
Space bar 11, 321
Space character 207, 208, 223
SPACES 223
SPECIAL 29, 112
Special Form Jump Table 382
Special forms 185, 362
SPECIAL-FORM-P 114
Speed improvements 352
SPLIT 98
Spread function 189
SQRT 287
Stable list merge 99
Stable sort 100
STABLE-SORT 100
Stack 106
Stack segment 361
Stack space 31
STACK-LIST 69
Standard input file 250
Standard input source 206, 250
Standard output file 255
Standard output sink 220, 255
Starting muLISP 13
STATS.LSP 6
Status line 320
String functions 158
String space 31, 203
STRING-CAPITALIZE 165
STRING-DOWNCASE 165
STRING-EQUAL 161
STRING-GREATERP 163
STRING-LEFT-TRIM 164
STRING-LESSP 163
STRING-NOT-EQUAL 163
STRING-NOT-GREATERP 163
STRING-NOT-LESSP 163
STRING-RIGHT-TRIM 164
STRING-TRIM 164

STRING-UPCASE 165
STRING/= 162
STRING< 162
STRING<= 162
STRING= 161
STRING> 162
STRING>= 162
STRINGP 40
STRUCTUR.LSP 6
Structures 4, 294
Sub-menu 321
SUB1 133
SUBLIS 97
SUBLIST 65
Subprocess invocation 3, 183
SUBSETP 93
SUBST 87
SUBST-IF 88
SUBSTITUTE 85
SUBSTITUTE-IF 86
Substitution functions 85
SUBSTRING 160
Super-parenthesis 216
Symbol creation 168
Symbol list 69
SYMBOL-FUNCTION 113
SYMBOL-PLIST 109
SYMBOL-VALUE 101
SYMBOLP 40
Symbols 28, 101
SYMEVAL-IN-CLOSURE 307
Syntax Error 36, 216
SYS file 266
SYSTEM 182
System calls 246
System option 33, 274

T 29, 40
Tab character 207, 208
Tab key 11, 16, 321, 327, 329
Tab stop 225, 327
Table of Contents v
TAILP 48
TAN 290
Tangent 290
TANH 292
TCONC 76
TENTH 56
Terminating macro characters 214
TERPRI 223
Text editor 317
Text entry mode 320
THIRD 56
Thrashing 32, 201
THROW 181
TIME 238
Token 15, 208
TONE 240, 384
Top-level loop 269
Top-level option 33, 274
Toshiba T3100 computer 371
Tower of Hanoi puzzle 5, 24
Trace mode 335, 337
Trace option 336, 337
TraceIn option 336
Trademark, registered ii
Transfer Clear option 324
Transfer Delete option 324
Transfer Load option 324
Transfer Merge option 324
Transfer Save option 324
TREE-EQUAL 47
True 40, 50
TRUNCATE 141
TURTLE 312
Turtle graphics 309
TURTLE-WINDOW 312
Tutorial system 5, 7, 25

Unclaimed Message 36, 303
Undefined Function 36, 170, 185, 190, 276
Undefined Structure 37, 296
UNDEFINED-FUNCTION 276
UNDEFMETHOD 300
Undelete option 324
UNDERFLOW 124
Underline 373
UNION 89

Universal time 239
UNLESS 173
Unnamed record structure 294
Unnamed Structure 37, 295
UNPACK 158
UNREAD-CHAR 210
Unsupported Option 37, 294
Untrace All option 337
Untrace One option 337
UNWIND-PROTECT 180
Up arrow key 11, 327
UPDATE-STATE 343
UPDATE-WINDOW 340
UPPER-CASE-P 151
Uppercase characters 151
User interface 269, 346
User querying 278

Value 101
Value cell 28
Value functions 28
Variable stack 31, 69, 362, 363
VECTOR 282
Vector cell 29
Vector space 31
Vectors 282
Version date 9, 13
Version number 9, 13
Vertical-bar character 207
VGA graphics adapter 8, 371
Video graphics adapter (VGA) 8
Video modes 233, 371
VIDEO-MODE 233
Visual display page 235

WHEN 173
Whitespace characters 207, 208
Whoppers 297
Wild-card characters 264
Window Close option 325
Window Designate option 325
Window Flip option 325
Window Goto option 325
Window manager 3, 339
Window Next option 325
Window Open option 321, 325, 335
Window option 337, 341
Window Previous option 325
Window Split Horizontal option 325
Window Split option 335
Window Split Vertical option 325
WINDOWS 343
WINDOWS.LSP 6, 318, 339
Window size limit 387
Word 327
WordStar-like editor 318, 326, 332, 378
Work window 320
Working copy of muLISP 9
WRITE 30, 220
Write protecting 9
WRITE-BYTE 224
WRITE-LINE 222
WRITE-STRING 222

Y-OR-N-P 278
YES-OR-NO-P 278

ZAP-STRING 203
Zapping print names 203
Zero Divide 37, 121
ZEROP 125

AUTO-NEWLINE 225
BLANK-COMPRESS 225
BREAK-LEVEL 271
COMMAND-LINE 340
CONDENSE 120
EXACT-MODE 122
FREE-LIST 203
GCHOOK 202
GCSTAT 201, 388
GENSYM-COUNT 168
GENSYM-PREFIX 168
IGNORE-CASE 219
INIT-WINDOW 343
INPUT-ECHO 218
INPUT-FILE 253
INSERT-MODE 280

INTERRUPT-HOOK 34, 276
LEFT-BUTTON 315
LINE-COLUMN 279
LINE-POINT 279
LINE-TERMINATOR 279, 387, 388
LOAD-VERBOSE 268
MOUSE-X 314
MOUSE-Y 314
OUTPUT-ECHO 225
OUTPUT-FILE 259
PLOT-COLOR 237
PRINT-BASE 227
PRINT-DOWNCASE 226
PRINT-ESCAPE 226, 385
PRINT-POINT 228
PRINTER-ECHO 225
PROMPT 342
READ-BASE 218
READ-STRINGS 212
READ-UPCASE 218
RIGHT-BUTTON 315
STRING-INDEX 212, 213
TAB-EXPAND 218
THROW-TAG 181
TIME-ZONE 239
UNDEFINED-HOOK 276

+TRAP 277
-TRAP 277
*TRAP 277
/TRAP 277

= 127
/= 127
< 128
> 128
<= 129
>= 129

, 216
' 216
(216
) 216
; 217
] 216
` 217
\ 19, 20
$ 14, 271

* 132, 271
** 271
*** 271
+ 131, 271
++ 271
+++ 271
- 131, 271
/ 132

1. Introduction
2. Fundamentals
3. Learning muLISP
4. Interpreter
5.1 Predicates
5:2 Lists
5.3 Symbols
5.4 Numbers
5.5 Print Names
5.6 Control Constructs
5.7 Interpreter
5.8 Communication
5.9 File System
5.10 User Interface
6.1 Arrays
6.2 Irrational Functions
6.3 Structures
6.4 Flavors
6.5 Closures
6.6 Turtle Graphics
6.7 Mouse Interfaces
7. Program Development System
8. Compiler
9. Machine-coded Routines
10. LISP References

LISP Language Programming Environment

muLISP-90 is a high performance LISP system for your MS-DOS based personal computer. *muLISP* provides:

- a full-featured pseudo-code interpreter
- a high performance native-code compiler
- an integrated windowing programming environment
- the ideal delivery vehicle for your AI application

System requirements: An IBM PC compatible or other personal computer running MS-DOS version 2.1 or later, 256K memory, and a 5¼ inch (360K) or a 3½ inch (720K) diskette drive. muLISP is also available for NEC PC-9801 compatible computers with a 5¼ inch (640K) or a 3½ inch (720K) diskette drive.

muLISP 90 TM

1. Introduction
2. Fundamentals
3. Learning muLISP
4. Interpreter
5.1 Predicates
5.2 Lists
5.3 Symbols
5.4 Numbers
5.5 Print Names
5.6 Control Constructs
5.7 Interprter
5.8 Communication
5.9 File System
5.10 User Interface
6.1 Arrays
6.2 Irrational Functions
6.3 Structures
6.4 Flavors
6.5 Closures
6.6 Turtle Graphics
6.7 Mouse Interfaces
7. Program Development System
8. Compiler
9. Machine-coded Routines
10. LISP References

LISP
Language
Programming
Environment

muLISP-90 is a high performance LISP system for your MS-DOS based personal computer. *muLISP* provides:

- a full-featured pseudo-code interpreter
- a high performance native-code compiler
- an integrated windowing programming environment
- the ideal delivery vehicle for your AI application

System requirements: An IBM PC compatible or other personal computer running MS-DOS version 2.1 or later, 256K memory, and a 5¼ inch (360K) or 3½ inch (720K) diskette drive. muLISP is also available for NEC PC-9801 compatible computers with a 5¼ inch (640K) or a 3½ inch (720K) diskette drive.

muLISP 90™